Comprehensive Java Programming

Building robust applications using core and advanced Java concepts

Dr. Saloni Bhushan

bpb

Comprehensive Java Programming

Building robust applications using core and advanced Java concepts

Dr. Saloni Bhushan

www.bpbonline.com

First Edition 2025

Copyright © BPB Publications, India

ISBN: 978-93-65890-112

All Rights Reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

To View Complete
BPB Publications Catalogue
Scan the QR Code:

www.bpbonline.com

Dedicated to

My beloved parents:
Shri Jata Shanker Prasad Singh
Smt. Neelam Singh
and
My husband Ravi *and daughters* ***Aanya*** *and* ***Lavanya***

About the Author

Dr. Saloni Bhushan is a dedicated academician and author with expertise in computer science. With an illustrious career spanning more than two decades, Dr. Bhushan has made significant contributions to the field through a seamless combination of teaching, research, and academic leadership. She has consistently inspired students and contributed to the academic community.

Dr. Bhushan's work reflects a perfect blend of theoretical knowledge and practical insights, making her a trusted guide for learners seeking to master Java programming. Through her teaching and writings, she strives to simplify complex concepts and ignite intellectual curiosity in her readers.

About the Reviewer

Satish Prahalad Gururajan is a passionate and highly skilled Solutions Architect with nearly two decades of professional experience in the tech industry. He has a strong foundation in application development, with a focus on Java technologies, and has played pivotal roles in software design, development, and delivery. Over the course of his career, Satish has excelled in diverse positions, ranging from developer to architect, product owner, and designer, which has allowed him to build a comprehensive understanding of technical and business needs.

Satish's expertise goes beyond development; he has consistently demonstrated a strong ability to bridge the gap between technical teams and business stakeholders, ensuring that solutions align with both strategic goals and user requirements. His experience spans a variety of industries, and he has a proven track record of successfully leading teams through the development lifecycle, from ideation and design to implementation and deployment.

As a certified professional, Satish holds multiple industry-recognized certifications, including SAFe® Product Owner/Product Manager, **Certified Scrum Product Owner** (**CSPO®**), and **Certified Scrum Master** (**CSM®**), which have honed his skills in agile methodologies and product management. These certifications have not only enriched his technical acumen but also enhanced his leadership and project management capabilities, enabling him to guide teams in delivering high-quality software in a collaborative and efficient manner.

Satish is especially passionate about problem-solving and has a natural talent for distilling complex technical concepts into easily understandable terms. His ability to simplify intricate topics allows him to communicate effectively with technical and non-technical audiences.

Acknowledgement

I would like to take a moment to acknowledge those who have been the motivation and continuous support during the writing of this book. First and foremost, I am deeply thankful to my parents for instilling in me the values of ambition and perseverance, and teaching me to set goals and work diligently to achieve them. Their guidance has been the foundation of my journey.

To my husband, Ravi, and my daughter, Lavanya, who has been my closest advisor, continuously encouraged me to write the book. To my younger one, thank you for your patience and understanding; this book would not have been possible without your love and support.

To my incredible in-laws, Mrs. Rani Singh, and my mentor, Dr. Bibhuti Bhushan Prasad Singh, who has been instrumental in shaping my career. My family, whose love and care are the bedrock of my existence.

A special mention goes to my younger sister, who always keeps me on my toes, pushing me to reach greater heights.

I am also immensely grateful to Satish Prahalad Gururajan, whose professional expertise and valuable suggestions have significantly contributed to the quality and completeness of this book. Your insights have been truly invaluable.

My sincere thanks to the team at BPB Publications for their unwavering support and commitment throughout this journey. Your professionalism and encouragement have made this process smooth and rewarding.

Finally, to you, the reader—thank you for choosing this book. Your passion for learning and growth is what motivated me to embark on this project. I hope this book serves as a valuable resource in your journey to mastering Java.

Preface

The evolution of technology has underscored the significance of programming languages in solving complex problems, driving innovation, and building robust applications. Among these languages, Java stands out for its versatility, reliability, and widespread adoption across diverse industries. This book, Comprehensive Java Programming: From Basics to Advanced Web Development, is an earnest effort to guide readers through the intricacies of Java programming, from fundamental concepts to advanced techniques in web development.

The primary goal of this book is to bridge the gap between theoretical knowledge and practical application. It is structured to cater to a broad audience, including students, educators, and aspiring developers, offering a clear, concise, and methodical approach to mastering Java. Beginning with foundational principles, readers will journey through advanced topics, including graphical user interface development, database integration, and enterprise-level frameworks like Spring and Hibernate.

The book also delves into web application development using servlets, JSP, and JSF, empowering learners to build dynamic and scalable solutions for modern web environments. Each chapter is crafted with illustrative examples, hands-on exercises, and case studies to foster active learning and problem-solving skills.

In writing this book, I drew from my years of teaching, research, and software development experience. I am confident that the blend of theoretical depth and practical insights presented here will enable readers to not only understand Java but also apply it effectively to real-world scenarios.

I hope this book serves as a valuable resource in your journey to becoming a proficient Java developer. Your feedback and suggestions are always welcome, as they will inspire me to continue improving and contributing to the field of computer science.

This book is divided into 15 chapters. The first part of the book introduces the fundamentals of Java development. The second part of the book covers advanced Java development concepts like GUI building, Database integration, web development, and different frameworks. The details are listed below.

Chapter 1: Introduction to Java Programming- is a foundational introduction to the Java programming language, aiming to equip readers with essential knowledge of its history, key features, and architecture. Additionally, it explores the practical aspects of Java development, including the role of **Integrated Development Environments** (**IDEs**) and **the Java Development Kit** (**JDK**).

Chapter 2: Flow Control and Arrays- will build upon the understanding of Java syntax by exploring control flow structures, such as conditional statements and loops, essential for directing program execution. Readers will also delve into the concept of arrays, a fundamental data structure for storing and manipulating collections of elements.

Chapter 3: Building Classes- introduces the concept of classes, the building blocks of object-oriented programming in Java. Readers will learn how to encapsulate data and behavior within classes, along with exploring access modifiers and method overloading to control class accessibility and functionality.

Chapter 4: Inheriting Behavior- explores inheritance, a core principle of object-oriented programming. Readers will understand how classes can inherit properties and behaviors from parent classes, facilitating code reuse and promoting modular design. Topics such as method overriding and abstract classes further enhance their understanding of inheritance.

Chapter 5: Interface Abstraction and Packaging- expands on the concept of abstraction through interfaces, providing a mechanism for defining contracts that classes can implement. Additionally, readers will explore the use of packages to organize and modularize code, promoting reusability and maintainability in larger Java projects.

Chapter 6: Exception Handling- explore how to handle exceptional situations in Java programs through exception-handling mechanisms. Understanding how to effectively manage errors and

unexpected conditions is crucial for writing robust and reliable Java applications. Topics such as try-catch blocks and best practices in exception handling are covered to ensure comprehensive error management.

Chapter 7: Input/Output Operations- introduces readers to **input/output** (**I/O**) operations in Java, enabling interaction with external data sources and storage. Understanding I/O operations is essential for building applications that read from and write to files, streams, and other data sources. Topics include byte and character streams, file I/O, and automatic file closing mechanisms.

Chapter 8: Concurrent Programming- covers the concept of concurrent programming and explores multithreading in Java, which allows programs to execute multiple tasks concurrently. Readers will learn how to create and manage threads, synchronize access to shared resources, and coordinate thread execution, enhancing the responsiveness and efficiency of Java applications.

Chapter 9: Swing GUI Fundamentals- explore Java's Swing framework for building **graphical user interfaces** (**GUIs**). Building upon their understanding of classes and event handling, they explore Swing components, layout managers, and event listeners to create interactive and visually appealing Java applications.

Chapter 10: Database Connectivity with JDBC- introduces readers to **Java Database Connectivity** (**JDBC**), enabling Java applications to interact with relational databases. By integrating JDBC with concepts from previous chapters, readers learn how to establish database connections, execute SQL queries, and process query results, laying the groundwork for database-driven Java applications.

Chapter 11: Advanced Networking in Java- explores advanced networking topics in Java. Readers dive into socket programming, URL handling, and TCP/IP communication, enabling them to build distributed and networked applications with Java.

Chapter 12: Web Component Development with Servlets- explores Java EE web components for developing robust web applications. Readers delve into servlet architecture, request handling, and state management mechanisms, enabling them to build scalable and dynamic web applications in Java.

Chapter 13: Dynamic Web Content with JSP- extends web application development capabilities by introducing **JavaServer Pages** (**JSP**). Readers explore JSP architecture, lifecycle, and integration with servlets, empowering them to create dynamic web content using Java-based technologies.

Chapter 14: Modern Web Development with JSF- focuses on web development with exploring the **JavaServer Faces** (**JSF**) framework. By leveraging JSF components, lifecycle phases, and configuration, readers learn to build modern and interactive web applications in Java.

Chapter 15: Java Beans and Web Frameworks- covers advanced Java web development concepts, including Java Beans and popular frameworks such as Spring MVC, Hibernate, and Struts. Readers explore architectural patterns, component lifecycles, and the comparative strengths of these frameworks, equipping them with the skills to architect and develop sophisticated web applications.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/c96fd8

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Comprehensive-Java-Programming**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.
Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

At **www.bpbonline.com**, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Table of Contents

1. Introduction to Java Programming

- Introduction
- Structure
- Objectives
- History of Java
- Features of Java
- Difference between Java and C++
- Java in today's software development
- Java architecture
  - *Java Development Kit*
  - *Java Virtual Machine*
  - *Java Runtime Environment*
- Java tokens: Keywords, Data types, literals, variables
  - *Keywords*
  - *Data types*
  - *Literals*
- Variables
  - *Rules for valid identifier*
- Scope and lifetime of variables
  - *Scope*
    - *Local scope*
    - *Class scope and instance variables*
    - *Class scope and static variables*
  - *Lifetime*
- Operators

Arithmetic operators
Assignment operators
Relational operators
Logical operators
Bitwise operators
Increment and decrement operator
Conditional (ternary) operator
Conclusion

2. Flow Control and Arrays
Introduction
Structure
Objectives
Basic Java program
Execution of the program
Structure of the program
One more simple program
Control structures
Conditional statements
if …else statement
Multiple if structure
If-else ladder
switch…case construct
Loops
for loop
while loop
do-while loop
Nesting of loops
Break, continue, and return statement
break
continue
return
Arrays

One-dimensional array
Declaration and initialization
Accessing elements
Length property
Iterating over elements
Multi-dimensional array
Declaring a multidimensional array
Initializing a multidimensional array
Example program
Conclusion
Exercises

3. Building Classes
Introduction
Structure
Objectives
Introducing class
Simple class example
Adding methods to class
Returning value from a method
Method with parameters
Constructors
Benefits of using a constructor
Comparison to methods
Object access modifiers
Private access modifier
Default access modifier
Protected access modifier
Public access modifier
Method overloading
Benefits of method overloading
Garbage collection
'this' keyword

Uses of 'this' keyword
'static' keyword
'static' variables
'static' methods
'static' blocks
'final' keyword
'final' variables
'final' methods
'final' classes
Wrapper classes
String class and methods
Conclusion
Exercises

4. Inheriting Behavior
Introduction
Structure
Objectives
Introducing inheritance
Types of inheritance
Single inheritance
Multilevel inheritance
Hierarchical inheritance
Hybrid inheritance
Multiple inheritance
'super' keyword
Accessing superclass methods
Accessing superclass fields
Calling superclass constructors
Method overriding
Key concepts and rules of method overriding
Benefits of method overriding
Dynamic method dispatch

Covariant return type
Role of final keyword in inheritance
Abstract class
Key characteristics of abstract classes
When to use abstract classes
Conclusion
Exercises

5. Interface Abstraction and Packaging
Introduction
Structure
Objectives
Interfaces
Defining an interface
Implementing an interface
Benefits of using interfaces
Abstract class vs. interface
Extending an interface
Packages
Defining a package
Importing a package
Package naming conventions
Conclusion
Exercises

6. Exception Handling
Introduction
Structure
Objectives
Exception class
Exception handling blocks
Handling Exceptions with try..catch
throw clause

throws
finally
Built-in checked and unchecked exceptions
User-defined exceptions
Best practices for exception handling in Java
Catch specific exceptions, not generic ones
Avoid swallowing exceptions
Use finally for cleanup
Log exceptions using a logging framework
Throw custom exceptions for specific errors
Document exceptions with Javadoc
Centralize error codes and descriptions
Fail fast and loud
Handle exceptions at the appropriate level
Use multi-catch blocks for related exceptions
Conclusion
Exercises

7. Input/Output Operations
Introduction
Structure
Objectives
Elementary concepts of input/output
Streams
Types of streams
Byte stream classes
Common byte stream classes and their methods
InputStream
OutputStream
DataOutputStream
DataInputStream
Character stream classes
Common character stream classes and their methods

Reader
InputStreamReader
FileReader
BufferedReader
Writer
FileWriter
PrintWriter
Standard input/output
Reading data from user
Reading bytes directly
Reading characters using InputStreamReader
Reading using BufferedReader
Reading using Scanner Class
Printing on console
Handling error messages in Java
File input /output
File input/output using byte stream
Reading from file using byte stream
Writing to file using byte stream
Copying a file using byte streams
File input/output using character streams
Reading characters from a file using FileReader
Writing characters to a file
Reading using BufferedReader
Writing lines to a file using BufferedWriter
Using PrintWriter for simplified writing
Automatic file closing
File reading using try-with-resources
Multiple resources in try-with-resources
Benefits of try-with-resources
Conclusion
Exercises
8. Concurrent Programming

Introduction
Structure
Objectives
Multithreading fundamentals
Thread class
Runnable interface
Understanding the main Thread
Creating a Thread
Extending the Thread class
By using Runnable interface
Creating multiple Thread
The life cycle of thread
Life cycle methods
Implementation of thread methods
Synchronization
Synchronized methods
Synchronized statements or blocks
Communication between Threads
Real-world examples
Conclusion
Exercises

9. Swing GUI Fundamentals
Introduction
Structure
Objectives
Containers
Top-level containers
JFrame
JWindow
JDialog
JApplet

Intermediate or general-purpose containers
JPanel
JScrollPane
JTabbedPane
JSplitPane
Components
JComponent
JLabel
ImageIcon
Buttons
JButton
JToggleButton
JCheckBox
JRadioButton
JTextField
JList
JComboBox
Layout managers in containers
FlowLayout
BorderLayout
GridLayout
CardLayout
GridBagLayout
Event handling
Common event classes and listener interfaces
Action event
Mouse event
Event Class: MouseEvent
Key events
Window events
Event adapters
Conclusion
Exercises

10. Database Connectivity with JDBC

Introduction

Structure

Objectives

Java Database Connectivity architecture

Advantages of JDBC architecture

Types of JDBC drivers

JDBC-ODBC bridge driver

Native-API Driver

Type 3: Network Protocol Driver

Thin Driver

Major JDBC classes and interface

DriverManager

Connection

Statement

PreparedStatement

CallableStatement

ResultSet

SQLException

Database operations using JDBC

Loading the JDBC driver

Establishing a connection

Creating a Statement

Executing SQL queries

Handling transactions

Handling ResultSet

Closing resources

Handling exceptions

Creating a simple JDBC application

Using prepared statement

Using CallableStatement

Exploring ResultSet operations

ResultSet concurrency constants

ResultSet holdability constants
JDBC with Swing
Mapping of Java-SQL data types
Using advanced data types
Binary Large Object
Character Large Object
Conclusion
Exercises

11. Advanced Networking in Java
Introduction
Structure
Objectives
Introduction to networking basics
IP address
Understanding ports
Client/server computing
Understanding socket
Mechanism of socket communication
Socket API
Socket types
Socket address
Working with URL and URLConnection
Creating a URL
Parsing a URL
Reading directly from a URL
URLConnection
InetAddress class
TCP/IP client sockets and server sockets
ServerSocket
Important methods in ServerSocket
Handling multiple clients
Socket class

Datagram communication with UDP
DataGramSocket class
DatagramPacket class
Example programs
Conclusion
Exercises

12. Web Component Development with Servlets
Introduction
Structure
Objectives
J2EE architecture
Single-tier architecture
Two-tier architecture
Three-tier architecture
N-tier architecture
Web application
Hypertext Transfer Protocol
Web containers
Servlets
Servlet life cycle
Destruction
Servlet API
HttpServlet
HttpServletRequest
HttpServletResponse
Writing service methods
Handling multi-valued parameter
State management
HttpSession
Cookies
Creating and sending cookies from the server

Modifying or deleting cookies
ServletContext
Example use cases of ServletContext
Sharing data across servlets
Sharing global configuration parameters
ServletConfig
Conclusion
Exercises

13. Dynamic Web Content with JSP
Introduction
Structure
Objectives
Introduction to JSP
Comparison with servlet
Code-structure and development style
Ease of use
Separation of concerns
Performance
Usage in modern web applications
JavaServer Pages architecture
Servlet container
JavaServer Pages life cycle
jspInit()
_jspService()
jspDestroy()
JSP standard syntax
JSP comments
JSP scripting elements
Java Server Pages directives
Include directive
Taglib directive
JavaServer Pages actions

JavaBeans with JSP
<jsp:useBean>
<jsp:setProperty>
<jsp:getProperty>
JSP implicit objects
request
response
out
session
application
config
Integration of JSP with servlet
JSP Standard Tag Libraries
JSP session management
HTTP sessions
Cookies in JSP
URL rewriting in JSP
Hidden form fields in JSP
Conclusion
Exercises

14. Modern Web Development with JSF
Introduction
Structure
Objectives
JavaServer Faces framework
Comparison with other web frameworks
JSF vs. Spring MVC
JSF vs. Struts
JSF vs. Angular
The architecture of JSF web applications
Model-View-Controller architecture
JSF UI components

Managed beans
Facelets view
FacesServlet
Navigation
Validation and conversion
Event handling
Managed bean scopes
Lifecycle phases of JSF request processing
Customizing the lifecycle
Building a JSF application
Creating a Maven project with JSF dependencies
Structure of a typical JSF web application
Project root directory
Source code directory (src/main/java)
Web resources directory (src/main/webapp)
Configuration and properties
Project structure summary
Configuration files
Conclusion
Exercises

15. Java Beans and Web Frameworks

Introduction
Structure
Objectives
Introduction to JavaBeans
Benefits of using JavaBeans
Creating a simple JavaBean: PersonBean
Using PersonBean in an application
Java Web Frameworks
Spring MVC Framework
Spring MVC architecture
Spring MVC module
Spring bean lifecycle

Spring API
Key annotations in the Spring API
Request handling workflow in Spring MVC
Form handling and data binding
Exception handling in Spring MVC
Advantages of using Spring MVC
Spring Boot and Spring MVC
Sample Spring MVC code
Struts Framework
Core components
Configuration files
Struts architecture
Struts Request Flow
Interceptors
Core interceptors in Struts
Custom interceptors
Interceptor Stack
Validator framework
Hibernate Framework
Hibernate architecture
Hibernate architecture layers
Hibernate mapping types
Collection mapping
Entity relationships mapping
Hibernate in web frameworks
Comparison between the frameworks
Conclusion

Index

Chapter 1
Introduction to Java Programming

Introduction

Java has been a cornerstone of the software industry for over 25 years, influencing the development of applications across a wide array of sectors, from finance to mobile app development. Initially released in 1995, Java's "write once, run anywhere" philosophy has enabled it to become a universal language, powering everything from enterprise-level applications to the Android ecosystem. In a world where technological landscapes constantly evolve, Java has not only endured but thrived, continuously adapting to meet the demands of modern software development.

In this chapter, we will explore the multifaceted nature of Java, beginning with its rich history and distinctive features that have allowed it to remain relevant in an ever-changing industry. We will study its architecture, including the **Java Development Kit** (**JDK**), **Java Virtual Machine** (**JVM**), and **Java Runtime Environment** (**JRE**), which together provide a robust framework for developers to write, compile, and execute code across diverse environments.

Furthermore, we will learn fundamental concepts such as Java tokens, data types, literals, variables, and operators. Understanding these core principles is essential for mastering Java programming and will serve as a foundation for more advanced topics. We will also discuss the scope and lifetime of variables, shedding light on the rules that govern their visibility and lifespan within Java applications.

To illustrate Java's relevance, we will examine its role in modern industries—how it supports large-scale systems in banking, enables the creation of dynamic web applications, and powers the Android mobile platform. This examination will not only highlight Java's versatility but also emphasize why it continues to be a preferred choice for developers worldwide.

Additionally, we will compare Java with C++, highlighting both their similarities and differences. This comparison will reveal Java's unique strengths and capabilities, providing insights into why this language remains a fundamental tool in software development today.

By the end of this chapter, readers will gain a comprehensive understanding of Java and its place in the technological landscape, enabling them with the knowledge to engage with this enduring language in new and innovative ways.

Structure

The chapter will cover the following topics:

- History of Java
- Features of Java
- Difference between Java and C++
- Java in today's software development
- Java architecture
- Java tokens: Keywords, data types, literals, variables
- Variables
- Scope and lifetime of variables
- Operators

Objectives

By the end of the chapter, the reader will be able to understand the history of Java, key features, distinctions from C++, its relevance in contemporary software development, and the fundamental components of Java architecture, including JDK, JVM, and JRE. Additionally, the chapter aims to explain Java tokens, such as data types, literals, and variables, along with insights into the scope and lifetime of variabls, and the usage of operators within the language.

History of Java

Java, initially conceived in the early 1990s by *James Gosling, Mike Sheridan, and Patrick Naughton* at *Sun Microsystems*, emerged officially in 1995 with its landmark release of Java 1.0. Originally named Oak, the language was rebranded as Java due to trademark issues. Java's standout feature was its platform independence, achieved through bytecode compilation, enabling the **write once, run anywhere** (**WORA**) mantra. This characteristic fueled Java's rapid adoption, particularly in web development and enterprise software. In 2006, Sun Microsystems open-sourced Java under the GPL, further propelling its popularity and fostering a vibrant open-source ecosystem. However, in 2010, Oracle Corporation acquired Sun Microsystems, becoming the new steward of Java. Under Oracle's management, Java continued to evolve, with a renewed focus on regular updates and a faster release cadence since 2017. Despite changes in ownership and management, Java has maintained its prominence, powering a diverse array of applications across different platforms and devices, solidifying its status as one of the most widely used programming languages globally.

Features of Java

Java has a rich set of features that contribute to its widespread adoption and enduring popularity among developers. Some of the key features of Java include:

- **Platform independence**: Java's WORA philosophy allows developers to write code once and run it on any platform that supports Java, thanks to its bytecode compilation and

the JVM.

- **Object-oriented**: Java is an object-oriented programming language, emphasizing the use of classes and objects to structure code and facilitate modular, reusable software development.
- **Simple and familiar syntax**: Java's syntax is derived from C and C++, making it familiar to developers with experience in those languages. Its straightforward syntax and strong readability simplify the learning curve for new programmers.
- **Automatic memory management**: Java incorporates automatic garbage collection, freeing developers from managing memory manually. The JVM automatically deallocates memory occupied by objects that are no longer in use, reducing the risk of memory leaks and improving application stability.
- **Robustness**: Java's strong type system, exception handling mechanisms, and runtime checking contribute to the language's robustness. These features help developers catch errors early in the development process and build reliable, fault-tolerant applications.
- **Security**: Java includes built-in security features, such as a robust security model, bytecode verification, and a security manager, which help protect against various security threats, such as unauthorized access and malicious code execution.
- **Portability**: Java's platform independence extends to its portability across different hardware architectures and operating systems. This portability allows Java applications to run seamlessly on diverse environments without modification.
- **Multi-threading support**: Java provides built-in support for multi-threading, allowing developers to create concurrent, scalable applications that can take advantage of modern multi-core processors.
- **Rich standard library**: Java comes with a comprehensive standard library (Java API), offering a wide range of classes

and methods for common tasks, such as I/O operations, networking, database connectivity, and GUI development.

- **Community and ecosystem**: Java benefits from a large and active developer community, supported by a vast ecosystem of libraries, frameworks, tools, and resources. This ecosystem fosters innovation and collaboration, enabling developers to leverage existing solutions and accelerate the development process.

These features collectively make Java a versatile, powerful, and widely adopted programming language, suitable for a diverse range of applications, from desktop and web development to mobile and enterprise solutions.

Difference between Java and C++

Java and C++ are both powerful programming languages with their own strengths and weaknesses. Here are some key differences between them:

	Java	
Memory management	Automatic garbage collection, where the JVM automatically deallocates memory for objects that are no longer referenced. This simplifies memory management and reduces the risk of memory leaks.	Manual memory management, where developers explicitly allocate and deallocate memory using pointers. This gives more control over memory usage but requires careful handling to avoid memory leaks and segmentation faults.
Platform independence	Platform-independent, as Java code is compiled into bytecode, which can run on any device or platform with a JVM installed, following the "write once, run anywhere" principle.	Platform-dependent, as C++ code is compiled into machine code specific to the target platform. Cross-platform development may require additional effort and platform-specific adaptations.
Syntax and features	Syntax is similar to C++, but with some differences and simplifications. It has a rich set of libraries and supports object-oriented programming, including concepts like classes, objects, inheritance, and polymorphism.	Syntax is more complex and allows for low-level memory manipulation, pointers, and operator overloading. It supports both procedural and object-oriented programming. It lacks certain C++ features like pointers and multiple inheritance for classes.
Performance	Generally slower than C++ due to its reliance on a virtual machine and automatic memory management. However, advancements like just-in-time (JIT) compilation have narrowed this gap.	Performs faster as it is closer to the hardware and allows for more efficient memory management. It is well-suited for performance-critical applications. However, advancements like JIT compilation have narrowed this gap.
Standard library	Comes with a powerful standard library (API) that includes classes and packages for various tasks such as I/O operations, networking, database connectivity, and GUI development. It offers a wide range of functionality, making it easier to develop applications quickly.	Has a standard library (STL) that includes classes and functions for various tasks, but it is not as extensive as Java's API. It may require additional third-party libraries or custom implementations for certain functionalities.
Concurrency support	Has built-in support for multithreading with features like the Thread class and synchronized methods, making it easier to write concurrent programs and take advantage of multi-core processors.	Lacks built-in support for multithreading, requiring the use of platform-specific libraries or third-party libraries. This can make concurrent programming more challenging and error-prone.

Table 1.1: Difference between Java & C++

Overall, the choice between Java and C++ depends on factors such as project requirements, performance considerations, and developer preferences. C++ is often favored for performance-critical and system-level programming, while Java is popular for its platform independence, simplicity, and robust ecosystem for enterprise and web development.

Java in today's software development

In today's software development landscape, Java continues to be a prominent and influential programming language, widely used in various domains and scenarios. Here is how Java remains relevant in contemporary software development:

- **Enterprise applications**: Java remains a top choice for building enterprise-grade applications due to its scalability, reliability, and robust ecosystem. Many large organizations rely on Java frameworks and technologies for mission-critical systems such as banking applications, **enterprise resource planning** (**ERP**) systems, and **customer relationship management** (**CRM**) software.
- **Web development**: Java is extensively used in web development, both for backend server-side logic and frontend user interfaces. Frameworks like Spring Boot, Jakarta EE (formerly Java EE), and Play Framework enable developers to build scalable and high-performance web applications. Additionally, JavaServer Pages (JSP) and **JavaServer Faces** (**JSF**) are popular choices for dynamic web content generation.
- **Mobile development (Android)**: Java is the primary programming language for developing Android applications. Android Studio, the official IDE for Android development, supports Java along with Kotlin, providing developers with a familiar and powerful language for building Android apps. While Kotlin has gained traction in recent years, Java remains a crucial part of the Android ecosystem.
- **Big data and analytics**: Java is widely used in big data processing and analytics applications. Frameworks like Apache Hadoop and Apache Spark, which are essential for

distributed computing and large-scale data processing, are implemented in Java. Additionally, Java's strong support for multi-threading and concurrency makes it suitable for parallel processing tasks in data-intensive applications.

- **Cloud computing**: Java is well-suited for developing cloud-native applications and microservices architectures. Platforms like Kubernetes, Docker, and OpenShift support Java applications, allowing developers to deploy and manage scalable, containerized services in cloud environments. Java's portability and platform independence make it a natural fit for cloud-based deployments.
- **Internet of Things (IoT)**: Java's lightweight runtime environments, such as Java Micro Edition (ME) and Java Embedded, are used in IoT devices and embedded systems. Java's portability, security features, and support for diverse hardware architectures make it a viable choice for developing applications in the IoT space.
- **Artificial intelligence and machine learning**: While Python is dominant in AI and machine learning, Java is still used in these fields, particularly for building enterprise-grade AI applications. Java libraries like *Deeplearning4j* and *Weka* provide tools for developing and deploying machine learning models, while frameworks like Apache Mahout offer scalable machine learning algorithms.

Overall, Java's versatility, reliability, and extensive ecosystem continue to make it a preferred choice for a wide range of software development projects, from enterprise applications to mobile apps and emerging technologies like IoT and cloud computing. As the language evolves with each new release, Java remains at the forefront of modern software development practices.

Java architecture

Java's architecture is based on the principle of WORA. It employs a multi-layered architecture that provides portability, security, and robustness. Java program goes through two processes: compilation and interpretation. Java compiler takes the source code and converts it into bytecode. This bytecode is portable and can be

ported to any JVM. The bytecode is then converted to machine code by JVM, which is executed by a machine. Understanding Java architecture involves grasping the roles of three key components: JDK, JVM, and JRE.

Java Development Kit

The JDK is a software development kit that provides tools for developing Java applications. It includes the **Java compiler** (**javac**), which translates Java source code into bytecode, and other development tools like Java disassembler (javap), Java documentation generator (javadoc), and jar (Java archive tool).

Additionally, the JDK contains the Java **Application Programming Interface** (**API**), which is a vast collection of classes and interfaces that developers use to build Java applications. These classes and interfaces provide functionality for tasks such as I/O operations, networking, database connectivity, and GUI development.

Java Virtual Machine

The JVM is an abstract computing machine that provides a runtime environment for executing Java bytecode. It acts as an intermediary between the compiled Java bytecode and the underlying hardware and operating system.

When a Java program is run, the JVM loads the bytecode, interprets it, and executes the corresponding machine instructions on the host system. Additionally, modern JVM implementations often use techniques like **just-in-time** (**JIT**) compilation to dynamically translate bytecode into native machine code for improved performance.

Java Runtime Environment

The JRE is a subset of the JDK that includes the JVM and essential libraries required to run Java applications. It does not contain development tools like the compiler and documentation generator.

Users need to install the JRE on their systems to run Java applications. It provides the necessary runtime environment for executing Java bytecode, ensuring platform independence and consistent behavior across different environments.

The JRE also includes platform-specific libraries and configurations that enable Java applications to interact with the underlying operating system, such as accessing files, networking, and managing system resources.

In summary, the role of different components of Java architecture can be defined as: the JDK provides tools and libraries for developing Java applications, the JVM executes Java bytecode on the target platform, and the JRE provides the runtime environment necessary for running Java applications. Together, these components form the foundation of Java's architecture, enabling developers to write platform-independent, scalable, and robust software solutions.

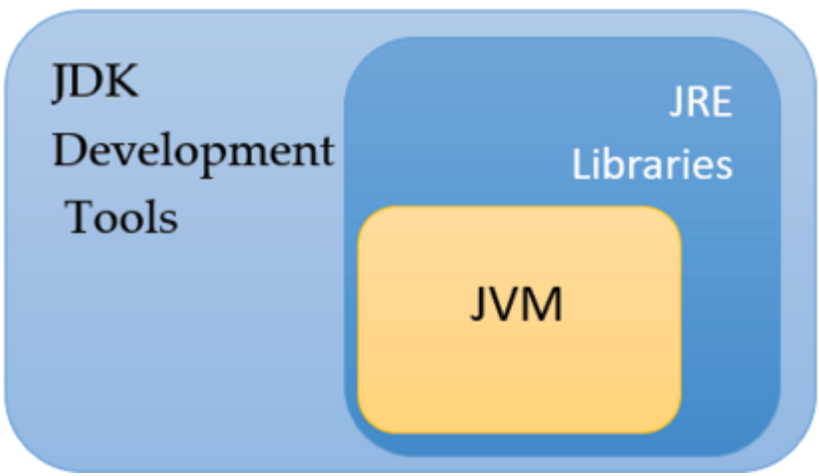


***Figure 1.1**: Java components*

Java tokens: Keywords, Data types, literals, variables

In Java, tokens are the smallest elements of a program that are meaningful to the compiler. They serve as the basic building blocks of the language syntax. Tokens include keywords (e.g., class, public, static), identifiers (names given to variables, methods, classes, etc.), literals (constant values like 100, 'a', "Hello"), operators (e.g., +, -, *, /, =), separators (e.g., (,), {, }, ;, .), and comments (e.g., // single-line comment, /* multi-line comment */). Each type of token

plays a distinct role in the structure and functionality of Java code, ensuring that the instructions written by developers are correctly interpreted and executed by the JVM. Understanding and correctly using these tokens is fundamental to writing syntactically correct and effective Java programs.

Keywords

Keywords are reserved words that have predefined meanings and are used to perform specific tasks or operations within the code. These words are part of the language syntax and serve as building blocks for constructing programs. As of the latest Java version, Java 17, there are 51 keywords. Here is a list of Java keywords:

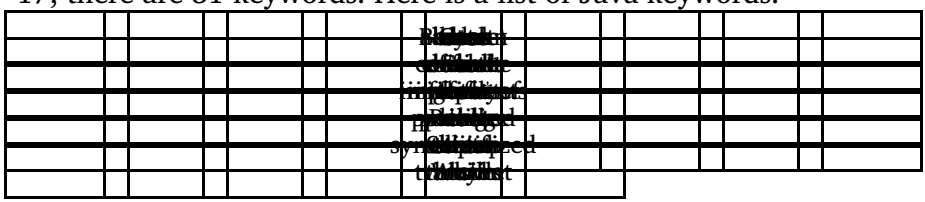


***Table 1.2**: List of Java keywords*

Note: const and goto are reserved for possible future use but are not currently used as keywords in Java.

Data types

Java supports various data types, including primitive data types and reference data types. Primitive data types are predefined by the language and represent basic values. They include:

- **int**: Represents integer values.
- **double**: Represents floating-point numbers.
- **Boolean**: Represents Boolean values (true or false).
- **char**: Represents single characters.
- **byte, short, long, and float**: Additional numeric types with different ranges and precision.

Reference data types are derived from classes and include:

- **String**: Represents a sequence of characters.
- **User-defined classes**: Created by the programmer to represent custom data structures.

Literals

Literals are constant values that are directly written into the code.

Examples of literals in Java include:

- **Integer literals**: 10, -5, 0x1A (hexadecimal), 0b1010 (binary).
- **Floating-point literals**: 3.14, -0.5, 1.0e-5 (scientific notation).
- **Character literals**: 'A', '%', '\n' (newline).
- **Boolean literals**: true, false.
- **String literals**: "Hello", "Java", "123".

Additionally, there are special literals like null, which represent the absence of a value.

Variables

Variables are named storage locations in memory used to hold data during program execution. Variable declaration in Java typically includes specifying the data type and the variable name. The syntax of a variable declaration is as follows:

data_type variable_name;

data_type can be any valid java data type, and **variable_name** should be any valid identifier. The declaration statement ends with semicolon (;), which is called a **statement terminator**.

Rules for valid identifier

The identifier name must begin with a letter, currency character ($), or underscore (_). It cannot start with a digit. After the first character, identifiers can include letters, digits, currency characters, or underscores. No spaces or special characters are allowed. Java is case-sensitive. So, uppercase and lowercase letters are considered different. For example, **myVar** and **myvar** are two distinct identifiers. Reserved words cannot be used as identifiers. Java has a set of reserved words, also known as **keywords**, which are part of the language syntax and have special meanings. Examples include int, class, if, else, etc.

CamelCase is a convention used in Java for naming identifiers,

especially for variables, methods, and sometimes class names. In CamelCase, the first letter of each word is capitalized except for the first word, which starts with a lowercase letter. CamelCase is not a strict rule enforced by the Java language itself but rather a widely adopted naming convention in the Java community. It helps improve code readability and consistency, making it easier for developers to understand and maintain codebases. It is especially useful when working with long or complex identifiers.

Here are some examples of valid identifiers:

myVar,
_variable, $price,
Test123,
totalAmount.

Here are some examples of invalid identifiers:

2ndVar (starts with a digit),
my Var (contains a space),
if (a reserved word),
my-variable (contains a hyphen),
my@var (contains a special character other than $ or _),

Examples of variable declarations:

- **int count;** //Declares an integer variable named count.
- **double temperature;** //Declares a double-precision floating-point variable named temperature.
- **boolean isReady;** //Declares a boolean variable named isReady.
- **String message;** // Declares a string variable named message.

Variables can be assigned values using the assignment operator (=):

- **count = 10;** //Assigns the value 10 to the variable count.
- **temperature = 98.6;** //Assigns the value 98.6 to the variable temperature.
- **isReady = true;** //Assigns the value true to the variable isReady.
- **message = "Hello, world!";** //Assigns the string "Hello, world!" to the variable message.

These tokens form the building blocks of Java programs, allowing developers to define data types, represent values, and store data in variables for processing and manipulation within their applications.

Scope and lifetime of variables

The scope and lifetime of variables are crucial concepts that determine where variables can be accessed and how long they persist in memory. The scope of a variable refers to the region of the code where the variable is visible and can be used. The lifetime of a variable is the period during which it exists in memory. Local variables exist only during the execution of the block or method in which they are declared and are destroyed once the block or method completes. Instance variables exist as long as the object they belong to is in memory, while static variables persist for the lifetime of the program. Understanding these concepts helps in managing memory efficiently and avoiding common errors like using uninitialized variables or memory leaks.

Scope

Scope refers to the region of the program where a variable is accessible or visible.

In Java, variables can have different scopes based on where they are declared:

Local scope

Variables declared within a block of code (enclosed within curly braces {}) have local scope and are accessible only within that block. They are typically used for temporary storage and are destroyed when the block exits.

Example:

```
1. public class LocalVariableExample {
2.     public static void main(String[] args) {
3.         // Declaring and initializing a local variable
4.         int x = 10;
5.         // Printing the value of the local variable
6.         System.out.println("The value of x is: " + x);
7.         // This is another block of code
```

```
        {
        // This is another local variable within a different block
            int y = 20;
            System.out.println("The value of y is: " + y);
        }

// Attempting to access y here would result in a compilation error
// as y is out of scope
    }
}
```

In this example, **x** is a local variable declared within the main method, and its scope is limited to that method. Similarly, **y** is a local variable declared within a nested block, and its scope is limited to that block. Attempting to access **y** outside of its block would result in a compilation error because it is out of scope.

Class scope and instance variables

Variables declared at the class level (outside of any method) but within a class are instance variables. They are accessible to all methods and constructors within the class and have the lifespan of the object they belong to. Observe the following class:

```
public class Car {
    // Instance variables (class scope)
    String brand;
    String model;
    int year;

    // Constructor
    public Car(String brand, String model, int year) {
        this.brand = brand;
        this.model = model;
        this.year = year;
    }

    // Method to display car information
    public void displayInfo() {
        System.out.println("Brand: " + brand);
        System.out.println("Model: " + model);
        System.out.println("Year: " + year);
```

```
    }

    public static void main(String[] args) {
        // Creating objects of Car class
        Car car1 = new Car("Toyota", "Camry", 2022);
        Car car2 = new Car("Honda", "Accord", 2021);

        // Calling method to display car information
        System.out.println("Car 1 Information:");
        car1.displayInfo();

        System.out.println("\nCar 2 Information:");
        car2.displayInfo();
    }
}
```

In this example, **Car** is a class with three instance variables: **brand**, **model**, and **year**. These variables are declared within the class but outside of any method, constructor, or block, giving them class scope.

Each object created from the **Car** class (e.g., **car1** and **car2**) has its own set of instance variables, and changes made to these variables in one object do not affect the variables of other objects. The **displayInfo()** method is used to display the information about a car object. More about classes we will be learning in *Chapter 3, Building Classes* under *Introducing class*.

Class scope and static variables

Static variables are declared using the **static** keyword and have class-level scope. They are shared among all instances of the class and are accessible without creating an object of the class.

```
public class Counter {
    // Static variable (class scope)
    static int count = 0;

    // Method to increment the count
    public static void incrementCount() {
        count++;
    }
```

```
9.
10.     // Method to display the count
11.     public static void displayCount() {
12.         System.out.println("Count: " + count);
13.     }
14.
15.     public static void main(String[] args) {
16.         // Increment count multiple times
17.         incrementCount();
18.         incrementCount();
19.         incrementCount();
20.
21.         // Display the count
22.         displayCount();
23.     }
24. }
```

In this example, **Counter** is a class with a static variable **count**. Static variables are declared using the static keyword, and they have class scope, meaning there's only one copy of the variable shared among all instances of the class. The **incrementCount()** method increments the static variable **count**, and **displayCount()** method prints the value of count.

In the main method, we increment the count three times using **incrementCount()** method and then display the count using **displayCount()** method. Since **count** is a static variable, changes made to it are reflected across all instances of the class.

Lifetime

The lifetime of a variable in Java refers to the duration for which the variable exists in memory. Local variables, declared within a method or block, have a lifetime limited to the execution of that method or block and are destroyed once it completes. Instance variables, declared at the class level but outside any method, have a lifetime that corresponds to the lifetime of the object they belong to; they are created when the object is instantiated and destroyed when the object is garbage collected. Static variables, also declared at the class level with the **static** keyword, have a lifetime that extends for the entire duration of the program, as they are created when the class is first loaded and destroyed when the program

terminates. Understanding the lifetime of variables is essential for effective memory management and avoiding issues such as memory leaks or unintentional data retention.

Example:

```
1. public class ScopeExample {
2.     // Instance variable
3.     int instanceVar;
4.
5.     // Static variable
6.     static int staticVar;
7.
8.     public void method() {
9.         // Local variable
10.        int localVar = 10;
11.        System.out.println(localVar); // Accessible within this method
12.        System.out.println(instanceVar); // Accessible within instance meth
13.        System.out.println(staticVar); // Accessible within instance method
14.    }
15.
16.    public static void main(String[] args) {
17.        ScopeExample obj = new ScopeExample();
18.        obj.method(); // Access instance and static variables
19.        // System.out.println(localVar); // Error: localVar is not accessible h
20.        // System.out.println(instanceVar); // Error: instanceVar is not static
21.        System.out.println(staticVar); // Access static variable
22.    }
23. }
```

Operators

Operators are symbols that perform operations on operands, such as variables, constants, or expressions. Java supports various types of operators, including arithmetic, assignment, relational, logical, bitwise, and conditional operators. Here is an overview of the different types of operators in Java:

Arithmetic operators

Arithmetic operators perform basic mathematical operations like addition, subtraction, multiplication, division, and modulus

(remainder). Here is a list of arithmetic operators:

***Table 1.3**: List of arithmetic operators*

In this table:

a = 10

b = 5

Here is one example demonstrating arithmetic operators:

```
1. public class ArithmeticOperatorsExample {
2.     public static void main(String[] args) {
3.         // Declare variables
4.         int a = 10;
5.         int b = 5;
6.
7.         // Addition (+)
8.         int addition = a + b;
9.         System.out.println("Addition (a + b): " + addition);
10.
11.        // Subtraction (-)
12.        int subtraction = a - b;
13.        System.out.println("Subtraction (a - b): " + subtraction);
14.
15.        // Multiplication (*)
16.        int multiplication = a * b;
17.        System.out.println("Multiplication (a * b): " +
    multiplication);
18.
19.        // Division (/)
20.        int division = a / b;
21.        System.out.println("Division (a / b): " + division);
22.
```

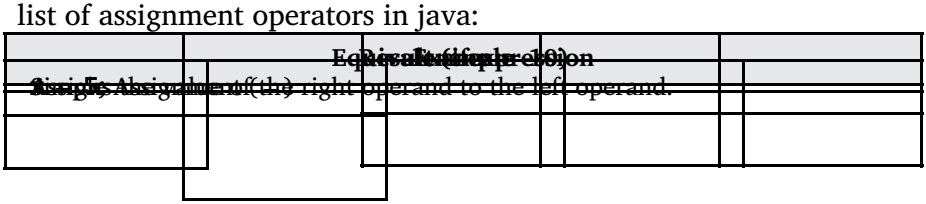
```
23.         // Modulus (%)
24.         int modulus = a % b;
25.         System.out.println("Modulus (a % b): " + modulus);
26.
27.         // Increment (++)
28.         a++; // Increment a by 1
29.         System.out.println("Increment (a++): " + a);
30.
31.         // Decrement (--)
32.         b--; // Decrement b by 1
33.         System.out.println("Decrement (b--): " + b);
34.
35.         // Unary Plus (+) (optional, indicates positive value)
36.         int positive = +a;
37.         System.out.println("Unary Plus (+a): " + positive);
38.
39.         // Unary Minus (-) (negating a value)
40.         int negative = -b;
41.         System.out.println("Unary Minus (-b): " + negative);
42.     }
43. }
```

The program will give following output:

Addition (a + b): 15
Subtraction (a - b): 5
Multiplication (a * b): 50
Division (a / b): 2
Modulus (a % b): 0
Increment (a++): 11
Decrement (b--): 4
Unary Plus (+a): 11
Unary Minus (-b): -4

Assignment operators

Assignment operators are used to assign values to variables. They can also combine arithmetic operations with assignments. Here is a list of assignment operators in java:



Addition Assignment (+=)	Adds the right operand to the left operand and assigns the result to the left operand.
Subtraction Assignment (-=)	Subtracts the right operand from the left operand and assigns the result to the left operand.
Multiplication Assignment (*=)	Multiplies the left operand by the right operand and assigns the result to the left operand.
Division Assignment (/=)	Divides the left operand by the right operand and assigns the result to the left operand.
Modulus Assignment (%=)	Takes the modulus of the left operand with the right operand and assigns the result to the left operand.

***Table 1.4**: List of assignment operators*

The following program demonstrates the use of assignment operators:

```
1. public class AssignmentOperatorsExample {
2.     public static void main(String[] args) {
3.         int x = 10;
4.
5.         System.out.println("Initial value of x: " + x);
6.
7.         // Addition assignment
8.         x += 5;  // Equivalent to: x = x + 5;
9.         System.out.println("After x += 5: " + x);  // x = 15
10.
11.        // Subtraction assignment
12.        x -= 3;  // Equivalent to: x = x - 3;
13.        System.out.println("After x -= 3: " + x);  // x = 12
14.
15.        // Multiplication assignment
16.        x *= 2;  // Equivalent to: x = x * 2;
17.        System.out.println("After x *= 2: " + x);  // x = 24
18.
19.        // Division assignment
20.        x /= 4;  // Equivalent to: x = x / 4;
21.        System.out.println("After x /= 4: " + x);  // x = 6
22.
23.        // Modulus assignment
24.        x %= 5;  // Equivalent to: x = x % 5;
25.        System.out.println("After x %= 5: " + x);  // x = 1
26.    }
27. }
```

The program will give the following output:

Initial value of x: 10
After x += 5: 15
After x -= 3: 12
After x *= 2: 24
After x /= 4: 6
After x %= 5: 1

Relational operators

Relational operators compare two values and return a Boolean result (true or false) based on the comparison. The relational operators are shown below:

***Table 1.5**: List of relational operators*

The following program demonstrates the relational operators:

```
1. public class RelationalOperatorsExample {
2.     public static void main(String[] args) {
3.         int a = 10;
4.         int b = 5;
5.
6.         // Equal to (==)
7.         System.out.println("a == b: " + (a == b));  // false
8.
9.         // Not equal to (!=)
10.        System.out.println("a != b: " + (a != b));  // true
11.
12.        // Greater than (>)
13.        System.out.println("a > b: " + (a > b));    // true
14.
15.        // Less than (<)
16.        System.out.println("a < b: " + (a < b));    // false
17.
18.        // Greater than or equal to (>=)
19.        System.out.println("a >= b: " + (a >= b));  // true
```

```
20.
21.         // Less than or equal to (<=)
22.         System.out.println("a <= b: " + (a <= b));  // false
23.     }
24. }
```

The program will give the following output:

a == b: false
a != b: true
a > b: true
a < b: false
a >= b: true
a <= b: false

Logical operators

Logical operators perform logical operations on Boolean values and return a Boolean result. They allow you to combine multiple conditions and evaluate them to produce a single Boolean result. The following are different logical operators:

	Result (if a = true, b = false)
Returns true if both operands are true; otherwise returns false.	Logical AND (&&)
Returns true if at least one operand is true; otherwise returns false.	Logical OR (||)
Reverses the logical state of its operand.	Logical NOT (!)

***Table 1.6**: List of logical operators*

The following program demonstrates the logical operators:

```
1. public class LogicalOperatorsExample {
2.     public static void main(String[] args) {
3.         boolean a = true;
4.         boolean b = false;
5.
6.         // Logical AND (&&)
7.         System.out.println("a && b: " + (a && b));  // false
8.
9.         // Logical OR (||)
10.        System.out.println("a || b: " + (a || b));  // true
11.
12.        // Logical NOT (!)
13.        System.out.println("!a: " + (!a));          // false
14.        System.out.println("!b: " + (!b));          // true
15.    }
```

```
16. }
17.
```

The program gives the following output:

```
a && b: false
a || b: true
!a: false
!b: true
```

Bitwise operators

Bitwise operators perform operations at the bit level on integer operands. They manipulate individual bits of operands. These operators are typically used in low-level programming, such as device drivers, embedded systems, and cryptography. Here is a list of bitwise operators in Java:

***Table 1.7**: List of bitwise operators*

The following program demonstrates bitwise operators:

```
1. public class BitwiseOperatorsExample {
2.     public static void main(String[] args) {
3.         int a = 5;  // 0101 in binary
4.         int b = 3;  // 0011 in binary
5.
6.         // Bitwise AND (&)
7.         System.out.println("a & b: " + (a & b));  // 0001 -> 1
8.
9.         // Bitwise OR (|)
10.        System.out.println("a | b: " + (a | b));  // 0111 -> 7
11.
12.        // Bitwise XOR (^)
13.        System.out.println("a ^ b: " + (a ^ b));  // 0110 -> 6
14.
15.        // Bitwise NOT (~)
```

```
        System.out.println("~a: " + (~a));       // 1010 -> -6 (two's complement representation)

        // Left Shift (<<)
        System.out.println("a << 1: " + (a << 1));  // 1010 -> 10

        // Right Shift (>>)
        System.out.println("a >> 1: " + (a >> 1));  // 0010 -> 2

        // Unsigned Right Shift (>>>)
        System.out.println("a >>> 1: " + (a >>> 1));  // 0010 -> 2
    }
}
```

In the above program, bitwise operators are used on two values, **5** and **3**, which are 0101 and 0011, respectively. In bitwise AND (&) both bits must be 1 for the result to be 1. So **a** & b results in 0001 (1). In bitwise OR (|), at least one bit must be 1 for the result to be 1. **a | b** results in 0111 (7). For bitwise XOR (^) the bits must differ for the result to be 1. a ^ **b** results in 0110 (6). Bitwise NOT (~) flips the bits. **~a** results in 1010 (which is -6 in two's complement). Left Shift (<<) shifts the bits of a to the left by 1 position (adds zeros from the right). **a << 1** results in 1010 (10). Right Shift (>>) shifts the bits of a to the right by 1 position.

a >> 1 results in 0010 (2). Unsigned Right Shift (>>>): Similar to >>, but shifts zeros from the left. **a >>> 1** results in 0010 (2).

The program gives the following output:

a & b: 1
a | b: 7
a ^ b: 6
~a: -6
a << 1: 10
a >> 1: 2
a >>> 1: 2

Example:

```
int a = 5; // Binary: 0101
```

```
2. int b = 3; // Binary: 0011
3. // Bitwise AND
4. int resultAnd = a & b; // resultAnd = 1 (Binary: 0001)
5. // Bitwise OR
6. int resultOr = a | b; // resultOr = 7 (Binary: 0111)
7. // Bitwise XOR
8. int resultXor = a ^ b; // resultXor = 6 (Binary: 0110)
9. // Bitwise NOT
10. int resultNotA = ~a; // resultNotA = -6
11. //(Binary: 1111 1111 1111 1111 1111 1111 1111 1010)
12.
13. // Left Shift
14. int resultLeftShift = a << 1;
15. // resultLeftShift = 10 (Binary: 1010)
16.
17. // Right Shift
18. int resultRightShift = a >> 1;
19. // resultRightShift = 2 (Binary: 0010)
20.
21. // Unsigned Right Shift
22. int unsignedRightShift = -10 >>> 1;
23. // unsignedRightShift = 2147483643 (Binary: 0111 1111 1111 1111 1
```

Let us go through each binary operation performed in the example step by step:

1. **Bitwise AND (a & b)**: Performed by statement 4 in the program

   a: 5 (Binary: 0101)

   b: 3 (Binary: 0011)

   Result: Performing bitwise AND operation on each corresponding pair of bits yields 0001 (which is 1 in decimal). This is because only the rightmost bit is set in both 'a' and 'b', resulting in 1.

1. **Bitwise OR (a | b)**: Performed by statement 6 in the program

   a: 5 (Binary: 0101)

   b: 3 (Binary: 0011)

   Result: Performing bitwise OR operation on each

corresponding pair of bits yields 0111 (which is 7 in decimal). This is because any bit that is set in either 'a' or 'b' results in a set bit in the result.

1. **Bitwise XOR (a ^ b)**: Performed by statement 8 in the program

   a: 5 (Binary: 0101)

   b: 3 (Binary: 0011)

   Result: Performing bitwise XOR operation on each corresponding pair of bits yields 0110 (which is 6 in decimal). This is because a bit is set in the result only if it is set in one operand but not both.

1. **Bitwise NOT (~a)**: Performed by statement 10 in the program

   The binary representation of the integer 5 is 0000 0000 0000 0000 0000 0000 0000 0101.

   When the bitwise NOT operator is applied to 5, it flips each bit, changing 0s to 1s and 1s to 0s.

   After applying the NOT operator, the binary representation becomes 1111 1111 1111 1111 1111 1111 1111 1010.

   Now, this binary representation can be interpreted as a negative number due to Java's two's complement representation of signed integers. In two's complement representation, the leftmost bit (most significant bit) indicates the sign of the number, where 0 represents a positive number and 1 represents a negative number.

   So, interpreting 1111 1111 1111 1111 1111 1111 1111 1010 as a signed integer, we have:

   The leftmost bit is 1, indicating a negative number.

   To find the decimal value, we can take the two's complement of the remaining bits and add 1.

   The two's complement of 111 1111 1111 1111 1111 1111 1010 is 000 0000 0000 0000 0000 0000 0101.

   Adding 1, we get 000 0000 0000 0000 0000 0000 0110.

   Result: Therefore, the result of applying the bitwise NOT

operator to the integer 5 is -6.

1. **Left Shift (a << 1)**: Performed by statement 14 in the program

   a: 5 (Binary: 0101)

   Result: Shifting the bits of 'a' one position to the left results in 1010 (which is 10 in decimal). This effectively multiplies 'a' by 2.

1. **Right Shift (a >> 1)**: Performed by statement 18 in the program

   a: 5 (Binary: 0101)

   Result: Shifting the bits of 'a' one position to the right results in 0010 (which is 2 in decimal). This effectively divides 'a' by 2.

1. **Unsigned Right Shift (-10 >>> 1)**: Performed by statement 22 in the program

   -10: In two's complement representation, -10 is represented as 11111111111111111111111111110110.

   Result: Shifting the bits of -10 one position to the right, filling the leftmost bit with 0, results in 01111111111111111111111111111011, which represents 2147483643 in decimal.

Increment and decrement operator

The increment (++) and decrement (--) operators are unary operators in Java used to increase or decrease the value of a variable by 1, respectively. They can be applied to numeric variables (int, float, double, etc.). Here is how they work:

- **Increment (++)**: Increases the value of the variable by 1.
- **Decrement (--)**: Decreases the value of the variable by 1.
- **They can be used in two forms**: postfix and prefix.
- **Postfix form**: The operator follows the operand.

```
1. int a = 5;
2. int b = a+
   +; // b will be assigned the value of 'a' (5), then 'a' will be incremen
3. // After this operation, 'a' will be 6 and 'b' will be 5
```

- **Prefix form**: The operator precedes the operand.

```
int a = 5;
int b = ++a; // 'a' will be incremented by 1 first, then assigned to 'b'
// After this operation, 'a' will be 6 and 'b' will also be 6
```

Similarly, the decrement operator works in the same fashion, just decrementing the value by 1.

Example:

```
int x = 10;
int y = 20;

// Postfix increment
int result1 = x++; // result1 = 10, x = 11

// Prefix increment
int result2 = ++y; // result2 = 21, y = 21

// Postfix decrement
int result3 = x--; // result3 = 11, x = 10

// Prefix decrement
int result4 = --y; // result4 = 20, y = 20
```

It is crucial to understand the difference between postfix and prefix forms, especially when the increment or decrement operation is part of a larger expression. In postfix form, the current value of the variable is used in the expression before it is incremented or decremented, while in prefix form, the variable is incremented or decremented before its value is used in the expression.

Conditional (ternary) operator

The conditional operator, also known as the ternary operator, is a concise way to express a conditional expression in Java. It is often used as a shorthand for simple if-else statements. The syntax of the conditional operator is as follows:

- condition ? expression1 : expression2
- The condition is evaluated first.
- If the condition is true, the expression1 is evaluated and returned.

- If the condition is false, the expression2 is evaluated and returned.

Example:

```
1. public class Main {
2.     public static void main(String[] args) {
3.         int a = 10, b = 20;
4.
5.         // Using the conditional (ternary) operator to find the maximum
6.         int max = (a > b) ? a : b;
7.
8.         System.out.println("The maximum value is: " + max);  // Outp
9.     }
10. }
```

The program will give the following output:

The maximum value is: 20

Conclusion

In this chapter, we covered Java's history, features, architecture, and fundamental concepts. The chapter also explores the differences between Java and C++, highlighting their respective strengths and use cases. This chapter also discussed the role of Java in today's software development. The discussion on Java tokens, including data types, literals, and variables, lays the foundation for understanding Java programming constructs, while insights into the scope and lifetime of variables help developers manage data effectively within their programs. Lastly, the chapter gives an overview of Java operators, demonstrating how to perform various operations and manipulations within Java programs.

In the next chapter, we are going to learn about program structure, flow control, and arrays.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 2
Flow Control and Arrays

Introduction

Java compiler, by default, executes the code from top to bottom. This is known as the sequential approach to programming. The statements in the code are executed along the sequence in which they appear. In this chapter, we will start by diving into the basics of a simple Java program, learning how to execute it, and understanding its structure. From there, we will progress to explore various control structures, including conditional statements like if-else and switch-case, as well as loops such as for, while, and do-while. These control structures are the building blocks of any Java program, enabling us to make decisions and iterate through code efficiently.

Sometimes, we need to alter the normal flow of execution within a program. Java provides three main jump statements, break, control, and return. We will learn how these statements can be used to manipulate the flow of execution. Additionally, we will discuss the concept of arrays, a fundamental data structure in Java. We will cover both one-dimensional and multi-dimensional arrays, learning

how to declare, initialize, and manipulate them in our programs.

Structure

The chapter covers the following topics:

- Basic java program
- One more simple program
- Control structures
- Break, continue, and return statement
- Arrays

Objectives

By the end of this chapter, readers will be able to understand the fundamentals of Java programming through a structured approach, covering the basics of writing and executing Java programs. This chapter aims to provide a solid understanding of program structure, including control structures such as conditional statements (if-else, multiple if, switch-case) and loops (for, while, do-while), along with concepts like break and continue statements for control flow manipulation. Additionally, the chapter deals with the concept of arrays in Java, covering both one-dimensional and multi-dimensional arrays.

Basic Java program

Let us start with a basic Java program with essential components such as a class definition, the **main** method to understand how to code in Java. The following is an example of a basic Java program that prints "**Hello, World!**" to the console, followed by a brief explanation of its key components.

```
1. /*A basic Hello World program
2. Save the file as HelloWorld.java*/
3. class HelloWorld {
4. //Program entry point
5.     public static void main(String[] args) {
6.         System.out.println("Hello, World!");
7.     }
```

```
8. }
```

Note: The descriptions use the standard JDK, which can be downloaded from the Oracle website, or use OpenJDK, which is an open source.

In Java, a program consists of one or more classes. All Java code must reside inside a class. The code must be stored in a file with the same name as given to your class in which **main()** method is written. The case used in naming the class and file name should also match because Java is a case-sensitive language. The file extension should be **.java**.

Execution of the program

The source code should be compiled by using **javac** compiler. On the command line, it can be written as follows:

```
1. c:\>javac HelloWorld.java
```

The **javac** compiler creates a bytecode version of the program and saves it in a file **HelloWorld.class**. This is an intermediate representation of the program, which can be ported to any JVM. To execute the program, the **java** command can be used as shown below:

```
1. c:\>java HelloWorld
```

We need to specify the name of the class which we want to execute with the **java** command. It searches for a file with the given name and **.class** extension and executes the code.

After execution, it will display the following output:

```
Hello, World!
```

Note: For java development, different Integrated Development Environments (IDEs) such as IntelliJ IDEA, Eclipse, NetBeans and many more are available, which provide comprehensive tools, offering features like code completion, syntax highlighting, debugging capabilities, and project management. These IDEs streamline the development process by providing a unified interface for writing, testing, and debugging Java code. Any of these IDEs can also be used.

Structure of the program

Let us have a closer look at the structure of our first program.

The first two lines of the program are comments:

```
1. /*A basic Hello World program
2. Save the file as HelloWorld.java*/
```

/* This is a multiline comment in Java. It begins with /* and ends with */. Everything between these symbols, including new lines and other characters, is considered a comment and is ignored by the compiler. Multiline comments are useful for longer explanations or comments that span multiple lines. In real application, this part specifies the logic or the process flow of that part of the program, which can be very helpful in understanding the program.

The next statement is:

```
class HelloWorld {
```

This line defines a class named **HelloWorld** by using the class keyword. In Java, every program consists of one or more classes. At this stage, we will consider that our program consists of only one class. The entire code must be enclosed within opening curly brace ({) and closing curly braces (}).

The class declaration line is followed by a single line comment:

```
//Program entry point
```

Anything after // on the same line is considered a comment and is ignored by the compiler. Single-line comments are typically used for brief explanations or annotations.

The next line of the code is:

```
public static void main(String[] args) {
```

Every Java program starts execution using the main method. It serves as the entry point of the program. **public static void main(String[] args)** declares the main method. **public** makes the method accessible from outside the class. **static** indicates that the method belongs to the class itself, not to instances of the class. **static** keyword allows the **main()** method to be called by Java virtual machine without creating any object of the class. void means that the method does not return any value.

Any value that a method requires to work upon is received as variables specified within the parenthesis, which follows the name of the method. These values are called parameters. If a method does

not require any parameter, then an empty parenthesis will follow the method name. In the main method, there is only one parameter named **String [] args,** which declares a parameter named **args**, as an array of strings. Arrays are collections of similar objects. String object can hold a set of characters. It is used to pass command-line arguments to the program while executing the program. The parenthesis of the method is followed by opening curly braces (**{**), which indicates the start of the body of the **main** method. All the code comprising the method should be enclosed within the opening curly brace and closing curly braces of the main method.

The method body in this program consists of a single statement:

```
System.out.println("Hello, World!");
```

This line prints the string **"Hello, World!"** to the console. **System.out** refers to the standard output stream, typically the console. **println** is a method of the **PrintStream** class used to print a string followed by a newline character. This statement ends with a semicolon. In Java, each statement typically ends with a semicolon. This includes variable declarations, method calls, assignment statements, loop statements, and many others.

The first closing curly brace (**}**) indicates the end of the **main** method, and the second closing curly marks the end of the class.

A program may have many classes. The classes may or may not contain main method. Java compiler can compile all those classes but cannot run a class without **main()**. Therefore, to launch a Java application, there should be at least one class with the **main** function.

One more simple program

One of the most fundamental concepts of a programming language is variable. The variable is a named memory location where we can store some value. The value of the variable can be used during the course of the program, and its value can be changed during the execution of the program. Let us see how variables can be declared and used in a program:

```
1. public class VariableDemo {
2.     public static void main(String[] args) {
3.         // Declare variables
```

```
        int number;
        double decimalNumber;
        char character;
        boolean isTrue;

        // Assign values to variables
        number = 10;
        decimalNumber = 3.14;
        character = 'A';
        isTrue = true;

        // Print the values of variables
        System.out.println("Integer number: " + number);
        System.out.println("Decimal number: " + decimalNumber);
        System.out.println("Character: " + character);
        System.out.println("Boolean value: " + isTrue);
    }
}
```

When you run this program, you will get the following output:

Integer number: 10
Decimal number: 3.14
Character: A
Boolean value: true

Let us break down the code.

The body of the **main** method contains a declaration of four variables:

```
        int number;
        double decimalNumber;
        char character;
        boolean isTrue;
```

The syntax of a variable declaration is as follows:

Data_type variable_name;

Here, **data_type** defines the type of variable being declared, and **variable_name** is the name of the variable which can be used to reference the variable. **int, double, char, boolean** are the default data types of java language, which can hold integer, floating point numbers, character, and Boolean values, respectively. All variables must be declared before they can be used. More than one variable

can be declared in a single statement as a comma separated list. It can be written as follows:

Data_type variable_name1, variable_name2, ….., variable_namen;

The next four lines are assignment statements:

```
number = 10;
decimalNumber = 3.14;
character = 'A';
isTrue = true;
```

Single equal to sign is the assignment operator. The values on the right-hand side of the expression are according to the data type of the variables, which is assigned to the respective variable names mentioned on the left-hand side of the assignment operator.

Variable declaration and initialization can be done in a single step. It can be written as:

// Single statement for declaration and assignment
data_type variable_name = value;

For example:

int age = 30;
// Multiple variable declaration and assignment in a single statement
int age = 30, numberOfChildren = 2, yearsOfExperience = 5;

The next four lines are for printing the values:

```
1.  // Print the values of variables
2.  System.out.println("Integer number: " + number);
3.  System.out.println("Decimal number: " + decimalNumber);
4.  System.out.println("Character: " + character);
5.  System.out.println("Boolean value: " + isTrue);
```

In these output statements '+' sign is used to concatenate the value of variables to the preceding string. Observe that the string message is enclosed within double quotes (""), which is called string literal, while the variable name is kept outside double quotes. Because we want to print the value of variables, everything enclosed within ("") will be printed as it is.

Control structures

Control structures in Java are constructs that enable programmers

to control the flow of execution within a program. They allow for making decisions, looping through code blocks, and executing statements conditionally. The main control structures are selection constructs and iteration constructs.

Selection constructs are used when the program needs to make decisions based on certain conditions. They allow the program to choose between different paths of execution. The primary selection construct is the "if" statement. It evaluates a condition and executes a block of code if the condition is true. There are variations of the "if" statement, such as "if-else" and "if-else if-else" (also known as "if-else ladder"), which provide branching behavior based on multiple conditions.

Conditional statements

Conditional statements belong to selection constructs. These are used when the program needs to make decisions based on certain conditions. They allow the program to choose between different paths of execution. These statements help control the flow of a program by evaluating expressions that result in true or false, determining which path the execution should take. The primary conditional statements in Java include if, else if, else, and switch. They enable programmers to build logic into their applications, making the software responsive to varying inputs and situations.

if ...else statement

Executes a block of code if a specified condition is true. Otherwise, executes an alternative block of code. The general syntax of an if-else block is as follows:

```
if (condition) {
    // code block to execute if condition is true
} else {
    // code block to execute if condition is false
}
```

Example:

```
public class BasicIfElse {
    public static void main(String[] args) {
        int number = 10;

```

```
        // Check if the number is greater than 0
        if (number > 0) {
            System.out.println("The number is positive.");
        } else {
            System.out.println("The number is not positive.");
        }
    }
}
```

Output:

The number is positive.

Multiple if structure

This contains a series of multiple if statements. Each condition is evaluated independently, regardless of the outcomes of the previous conditions. If one condition evaluates to true, the corresponding block of code associated with that if statement executes, and then the program moves on to the next statement:

```
if (condition1) {
    // Code block 1
}
if (condition2) {
    // Code block 2
}
if (condition3) {
    // Code block 3
}
// And so on...
```

In this scenario, each condition is checked sequentially, and multiple blocks of code may execute if multiple conditions are true.

Example:

```
public class MultipleIfExample {
    public static void main(String[] args) {
        int num = 10;

        // Checking conditions using multiple if statements
        if (num > 0) {
            System.out.println("Number is positive");
        }
```

```
        if (num % 2 == 0) {
            System.out.println("Number is even");
        }
        if (num % 5 == 0) {
            System.out.println("Number is divisible by 5");
        }
    }
}


```

Output:

Number is positive
Number is even
Number is divisible by 5
since all three conditions are true for the number 10.

If-else ladder

In an if-else ladder, multiple conditions are checked sequentially, but once a condition evaluates to true, only the corresponding block of code associated with that true condition executes. If none of the conditions are true, the code within the else block (if present) executes:

```
if (condition1) {
    // Code block 1
} else if (condition2) {
    // Code block 2
} else if (condition3) {
    // Code block 3
} else {
    // Default code block
}

```

In this scenario, the conditions are evaluated one by one from top to bottom. If **condition1** is true, *Code block 1* executes, and the rest of the ladder is skipped. If **condition1** is false and **condition2** is true, *Code block 2* executes, and so on. If none of the conditions are true, the code within the else block (if present) executes as the default behavior.

Example:

```
1. public class IfElseIfLadderExample {
2.     public static void main(String[] args) {
3.         int marks = 75;
4.
5.         if (marks >= 90) {
6.             System.out.println("Grade A");
7.         } else if (marks >= 80) {
8.             System.out.println("Grade B");
9.         } else if (marks >= 70) {
10.            System.out.println("Grade C");
11.        } else if (marks >= 60) {
12.            System.out.println("Grade D");
13.        } else {
14.            System.out.println("Grade F");
15.        }
16.    }
17. }
```

Output:

Grade C

switch…case construct

The switch case construct allows for a concise and efficient selection of code blocks based on the value of a variable. It is commonly used when there are multiple conditions to be evaluated against the same variable. It evaluates an expression and executes code blocks based on matching cases:

```
1. switch (expression) {
2.     case value1:
3.         // code block to execute if expression matches value1
4.         break;
5.     case value2:
6.         // code block to execute if expression matches value2
7.         break;
8.     default:
9.         // code block to execute if no cases match
10. }
```

Example:

```java
public class SwitchExample {
    public static void main(String[] args) {
        int dayOfWeek = 3;
        String dayName;

        switch (dayOfWeek) {
            case 1:
                dayName = "Monday";
                break;
            case 2:
                dayName = "Tuesday";
                break;
            case 3:
                dayName = "Wednesday";
                break;
            case 4:
                dayName = "Thursday";
                break;
            case 5:
                dayName = "Friday";
                break;
            case 6:
                dayName = "Saturday";
                break;
            case 7:
                dayName = "Sunday";
                break;
            default:
                dayName = "Invalid day";
                break;
        }

        System.out.println("Day of the week: " + dayName);
    }
}
```

Output:

Day of the week: Wednesday

The **switch** statement evaluates the value of **dayOfWeek** and executes the corresponding case. Each case statement represents a

possible value of the variable **dayOfWeek**. If the value matches the case, the code block associated with that case is executed. In this example, if **dayOfWeek** is 1, then the variable **dayName** is assigned the value "**Monday**", and the break statement terminates the switch statement. The default case is executed when none of the case values match the value of the variable being switched (**dayOfWeek** in this case). It acts as a fallback option. If **dayOfWeek** does not match any of the defined cases (1 to 7), **dayName** is assigned the value "**Invalid day**". The break statement ends the switch statement.

Loops

Iteration constructs, also known as loops, allow the program to execute a block of code repeatedly as long as a certain condition is true. There are different types of loops, such as "for", "while", and "do-while" loops.

for loop

The for loop allows you to execute a block of code repeatedly based on a specified condition:

```
1. for (initialization; condition; update) {
2.     // code block to execute repeatedly
3. }
```

It consists of three sections: initialization, termination condition, and update expression. These sections work together to control the execution of the loop, allowing you to iterate over a block of code repeatedly until a specified condition is met. The initialization section is where you initialize the loop control variable. This variable will be used to iterate through the loop. This section is executed only once, at the beginning of the loop. The termination condition is a Boolean expression that determines whether the loop should continue executing. As long as the condition evaluates true, the loop will continue executing. If the condition evaluates to false, the loop terminates, and the program proceeds to the next statement after the loop. The update expression is responsible for updating the loop control variable after each iteration of the loop.

Example:

```
1. public class ForLoopExample {
2.     public static void main(String[] args) {
```

```
3.      int num=5;
4.    // Loop from 1 to 5
5.      for (int counter = 1; counter <= 5; counter++) {
6.          System.out.println(num+"*"+counter
    +":" + num*counter);
7.      }
8.    }
9. }
```

Output:

5*1: 5
5*2: 10
5*3: 15
5*4: 20
5*5: 25

In the above program, a loop control variable counter is initialized with a value of **1**. The condition **counter <= 5:** specifies that as long as the value of the counter is less than or equal to **5**, the loop will continue executing. The update expression **counter++** increments the value of the counter by 1 after each iteration of the loop. Inside the loop, **System.out**.println prints out the multiplication table for the number 5. It multiplies the value of num (which is 5) by the current value of counter and prints the result along with some explanatory text.

Observe the **print** statement here. It uses a combination of string literals and variable name to display the desired output.

while loop

The while loop repeatedly executes a block of code as long as a specified condition is true. Its structure consists of four main components: initialization, condition checking, loop body execution, and update of the control variable. Firstly, before entering the loop, you typically initialize one or more variables that will serve as loop control variables. These variables establish the starting point for the loop. Next, the loop condition is evaluated. If the condition evaluates to true, the loop body is executed. If it evaluates to false, the loop is terminated, and program control is transferred to the next statement after the loop. Within the loop body, any valid Java code can be included. This can involve assignments, method calls,

or other control flow statements. After the loop body is executed, there should typically be a mechanism that changes the loop control variable in such a way that the loop condition eventually becomes false. This step prevents the loop from running indefinitely, ensuring that the loop terminates eventually. The loop condition is reevaluated after each iteration. If it still evaluates to true, the loop body is executed again, and this process repeats until the condition evaluates to false.

```
while (condition) {
    // code block to execute repeatedly
}
```

Example:

```
public class WhileLoopExample {
    public static void main(String[] args) {
        int count = 1;//initialization of loop control variable

        // Loop while count is less than or equal to 5
        while (count <= 5) {
            System.out.println("Count is: " + count);
            count++; // Increment count by 1
        }
    }
```

Output:

Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5

do-while loop

The do-while loop is a control flow statement that repeatedly executes a block of code until a specified condition evaluates to false. Unlike the while loop, which evaluates the condition before the loop body is executed, the do-while loop executes the loop body first and then evaluates the condition. This ensures that the loop body is executed at least once, even if the condition is initially false. The structure of the do-while loop consists of the **do** keyword followed by the loop body enclosed in curly braces **{}**, and then the

while keyword followed by the loop condition in parentheses (). The loop condition is evaluated after each iteration of the loop. During each iteration, the loop body is executed, and then the loop condition is evaluated. If the condition evaluates to true, the loop body is executed again, and the process repeats. If the condition evaluates to false, the loop terminates, and program control moves to the next statement after the loop.

The do-while loop is useful in situations where you want to execute a block of code at least once, regardless of the initial condition. The general syntax of do-while loop is as follows:

```
do {
    // code block to execute repeatedly
} while (condition);
```

Let us rewrite the above program using the do -while loop:

```
public class DoWhileLoopExample {
    public static void main(String[] args) {
        int count = 1;

        // Execute the loop body at least once
        do {
            System.out.println("Count is: " + count);
            count++; // Increment count by 1
        } while (count <= 5);
    }
}
```

Output:

Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5

Nesting of loops

Nesting loops refers to having one loop inside another loop. This is a common technique used in programming to perform repetitive tasks that require multiple levels of iteration. Here is an example of nesting loops in Java:

```java
public class NestedLoopsPattern {
    public static void main(String[] args) {
        int size = 5;

        // Outer loop to iterate over rows
        for (int i = 1; i <= size; i++) {
            // Inner loop to iterate over columns
            for (int j = 1; j <= i; j++) {
                System.out.print("* "); // Print asterisks
            }
            System.out.println(); // Move to the next line after printing each
        }
    }
}
```

Output:

```
*
**
***
****
*****
```

We initialize a variable size with the value **5**. This represents the number of rows in our pattern. We have an outer loop (for loop) that controls the number of rows. It starts with **i = 1** and continues until **i** reaches the value of size (5 in this case).

For each iteration of the outer loop, we print one row of the pattern. Inside the outer loop, we have another loop (for loop) that controls the number of asterisks printed in each row. It starts with **j = 1** and continues until **j** reaches the value of **i**. For each iteration of the inner loop, we print an asterisk (*) followed by a space. The number of asterisks printed in each row increases with each iteration of the outer loop. After printing the asterisks for each row using the inner loop, we move to the next line (**System.out.println()**) to start a new row. As a result of the nested loops, we get a pattern of asterisks where the number of asterisks increases in each row. The pattern resembles a triangle, with each row having one more asterisk than the previous row.

Break, continue, and return statement

When programming, controlling the flow of execution within loops and methods is essential for efficient and logical code. The **break, continue**, and **return** statements are powerful tools in this regard. The break statement is used to exit a loop prematurely, effectively breaking out of it when a certain condition is met. The continue statement, on the other hand, skips the current iteration of a loop and moves on to the next iteration, allowing you to bypass specific logic without stopping the entire loop. Finally, the return statement is used within methods to exit the method and optionally return a value, ending the method's execution and sending control back to the calling function. Understanding and utilizing these statements is crucial for writing clear, concise, and efficient code.

break

The break statement is used to terminate the execution of a loop or switch statement prematurely. When encountered within a loop or a switch statement, the break statement immediately exits the loop or switch statement, and program control moves to the statement immediately following the loop or switch:

- **Inside a loop**: When a **break** statement is encountered within a loop (e.g., for, while, or do-while), the loop is terminated immediately, and program control moves to the next statement after the loop. This allows you to exit the loop prematurely based on some condition.

  For example, searching for an element in an array:

```
int[] numbers = {1, 2, 3, 4, 5};
int target = 3;
boolean found = false;

for (int number : numbers) {
    if (number == target) {
        found = true;
        break; // Exit the loop early if the target is found
    }
}

if (found) {
    System.out.println("Target found in the array.");
```

```
} else {
    System.out.println("Target not found in the array.");
}
```

- **Inside a switch statement**: When a break statement is encountered within a **switch** statement, it causes the program to exit the **switch** block. Without the break statement, execution would continue to the next case, resulting in fall-through behavior. Each case in a switch statement typically ends with a break statement to prevent fall-through.

```
switch (dayOfWeek) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    // More cases...
}
```

continue

The continue statement is used to skip the rest of the current iteration of a loop (e.g., for, while, or do-while) and proceed to the next iteration. When encountered within a loop, the continue statement causes the program to immediately jump to the loop's update expression (for for loops) or loop condition (for while and do-while loops), skipping any remaining statements in the loop body.

Example: Skipping even numbers in a loop:

```
for (int i = 1; i <= 10; i++) {
    if (i % 2 == 0) {
        continue; // Skip even numbers
    }
    System.out.println(i);
}
```

return

The **return** statement is used to exit from a method and optionally

return a value back to the caller. When a **return** statement is encountered within a method, the method's execution is immediately terminated, and control is passed back to the code that called the method. The return value, if specified, is passed back to the calling code.

Exits the current method and optionally returns a value.

Example:

```
1. public int add(int a, int b) {
2.     int sum = a + b;
3.     return sum; // Return the sum of a and b
4. }
```

Alternatively, it can be written as:

```
1. public int sum(int a, int b) {
2.     return a + b; // Return the product of a and b
3. }
```

If a method does not return any value (i.e., it has a return type of **void**), the **return** statement can still be used to exit the method. However, in **void** methods, the **return** statement is typically used without specifying a value.

```
1. public void greet() {
2.     System.out.println("Hello, world!");
3.     return; // Exit the method (optional in void methods)
4. }
```

In this example, the greet method prints a greeting message to the console and then exits. The return statement is optional in void methods because the method does not return any value.

Arrays

An array is a data structure that allows you to store multiple values of the same type under a single variable name. It provides a convenient way to manage collections of data elements of the same type.

Arrays can be one-dimensional or multidimensional.

One-dimensional array

A one-dimensional array in Java is a data structure that allows you

to store a fixed-size sequence of elements, all of the same data type, in a linear form. Each element in the array is accessed using an index, with the first element at index 0, the second at index 1, and so on. This structure is particularly useful when you need to store and manage a collection of related data, such as a list of numbers, names, or objects. For example, a one-dimensional array can be used to store the scores of students in a class, enabling easy access, manipulation, and computation of statistics like average, maximum, and minimum values. Arrays are efficient for iterating through and performing operations on large sets of data, making them fundamental in tasks like sorting, searching, and aggregating data.

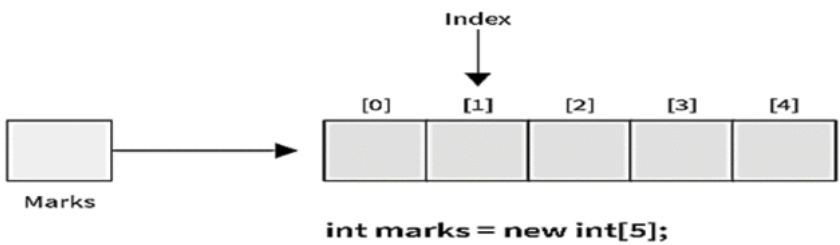


***Figure 2.1**: One dimensional array*

Declaration and initialization

Arrays are declared by specifying the type of elements they will contain, followed by square brackets [] and a variable name.

Example:

```
int[] numbers;
```

It declares an array of named numbers that can store integers.

Arrays can be initialized at the time of declaration or later using the **new** keyword along with the array size.

Example:

```
int[] numbers = new int[5];
```

It creates an array of integers with a length of **5**.

There is an alternative method of array declaration.

```
Data_type variable_name[]
```

In this example, the square bracket (**[]**) follows the variable name.

Example:

int numbers[] = new int[5]

This is equivalent to the above declaration.

The first alternative in which the type specifier is followed by **[]** is convenient while declaring several arrays in a single statement.

int [] a1, a2, a3; // declares three array

Array can be initialized at the time of declaration. The values can be supplied as a comma separated list enclosed within curly braces. In this case, specifying the size of array is not required. The array would be automatically created with a size to accommodate all values listed. new keyword is also not required. For example, to store the first ten prime numbers in an array, the following statement will create an integer array of 10 elements and assign the values to corresponding elements.

int[] primes = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};

Accessing elements

Elements in an array are accessed using an index, which represents the position of the element within the array. Array indices start from 0 and go up to length - 1, where length is the number of elements in the array.

Individual elements can be accessed using the square bracket notation [], followed by the index. For example,

int value = numbers[2]; //retrieves the third element from the array numbers

Length property

Every array in Java has a length property, which represents the number of elements it can hold. This property is accessed using dot notation. For example,

int size = numbers.length; //retrieves the length of the array numbers

Iterating over elements

Arrays can be traversed using loops such as for or foreach to access each element sequentially. For example:

```
for (int i = 0; i < numbers.length; i++) {
    System.out.println(numbers[i]);
}
```

Or,

```
for (int number : numbers) {
    System.out.println(number);
}
```

In Java, an enhanced **for** loop (also known as a for-each loop) is a simpler way to iterate over arrays or collections, compared to the traditional for loop. It eliminates the need to manually manage an index variable, making the code cleaner and less error-prone. The syntax of the loop is as follows:

```
for (type variable : collection) {
   // Body of the loop
}
```

Let us see one example program:

```
public class Main {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3, 4, 5};

        // Enhanced for loop
        for (int num : numbers) {
            System.out.println(num);
        }
    }
}
```

Multi-dimensional array

A multi-dimensional array in Java is an array of arrays, where each element is itself an array, allowing you to store data in a grid-like or matrix form. The most common type is a two-dimensional array, which can be visualized as a table with rows and columns. For instance, a two-dimensional array can be used to represent a matrix in mathematics, a chessboard in a game, or a table of student grades across different subjects. Multi-dimensional arrays are useful for managing and processing data that has more than one dimension, such as images (where you might have rows and columns of pixels),

or for performing operations in scientific computing, where multiple variables need to be stored and manipulated together. They allow for efficient organization and manipulation of data that inherently has multiple layers of information.

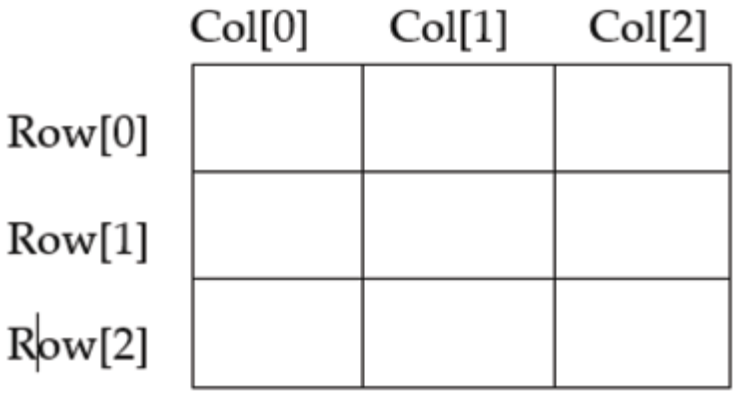


***Figure 2.2**: Two-dimensional array*

Declaring a multidimensional array

The general syntax of declaring a multidimensional array is:

dataType [] [] arrayName;

For example, for declaring an integer two-dimensional array, the statement would be as follows:

```
1. // Syntax: <type>[][] <arrayName>;
2. int[][] matrix; // Declares a 2D array named matrix.
```

Here, the brackets **[] []** indicate that the array will have two dimensions, typically representing rows and columns. Alternatively, it can be written as:

int matrix [] [] ;

This syntax achieves the same result but presents the array's name before the brackets. Both declarations are equivalent in functionality, with the choice between them largely depending on personal or stylistic preference. The first syntax is often favored for its clarity, as it explicitly separates the type from the variable, while the second syntax might be used to emphasize the variable name.

Initializing a multidimensional array

Initializing a multidimensional array in Java involves not only declaring the array but also allocating memory and, optionally, assigning initial values to its elements. A multidimensional array, often used to represent data in tabular or matrix form, requires specifying the size of each dimension during initialization. This can be done either during declaration or afterward, using different approaches such as inline initialization, where the array is populated with predefined values, or dynamic initialization, where elements are assigned individually or in a loop. Proper initialization ensures that the array is ready for use, allowing for efficient data storage and retrieval across its multiple dimensions. Understanding the various methods of initializing a multidimensional array is crucial for effectively managing complex data structures in Java. The general syntax for initializing a multidimensional array is as follows:

Syntax: **<arrayName> = new <type>[<rows>][<columns>];**

For example, the array named matrix declared above can be initialized with the following statement:

```
matrix = new int[3][3]; // Creates a 3x3 matrix of integers
```

A multidimensional array can be initialized by providing values for each dimension in its own set of curly braces:

```
int[][] matrix = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };
```

The above statement will create a two-dimensional array with 3 rows and 3 columns with the given values.

Accessing elements of a multidimensional array:

```
1. matrix[0]
   [0] = 1; // Assigns 1 to the element at row 0, column 0.
2. int value = matrix[1][2];
3. // Retrieves the element at row 1, column 2.
```

Iterating over elements of a multidimensional array:

```
1. for (int row = 0; row < matrix.length; row++) {
```

```
    for (int col = 0; col < matrix[row].length; col++) {
        System.out.println(matrix[row][col]);
    }
}
```

Example program

Let us see a basic Java program to demonstrate the use of a two-dimensional array. The program initializes a 2D array, populates it with values, and then prints out the array in a matrix form.

```
public class TwoDimensionalArrayDemo {
    public static void main(String[] args) {
        // Declare and initialize a 2D array (3x3 matrix)
        int[][] matrix = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        // Print the 2D array in a matrix form
        System.out.println("The 2D array (matrix) is:");
        for (int i = 0; i < matrix.length; i++) {       // Loop through rows
            for (int j = 0; j < matrix[i].length; j++) { // Loop through columns
                System.out.print(matrix[i][j] + " ");
            }
            System.out.println(); // Move to the next line after each row
        }
    }
}

```

Conclusion

In this chapter, we have learned the basic structure of a java program followed by different Control structures which enable programmers to dictate the flow of execution within a program. Key control structures in Java include conditional statements like if...else and switch...case, as well as looping structures like for,

while, and do-while loops. We have also covered break and continue statements, which help in altering the flow of control within loops and switch statements. Additionally, we discussed the return statement, which terminates the execution of a method and, optionally, returns a value to the calling code. We have also covered the concept of Arrays, the fundamental data structures in Java, allowing for the storage of multiple values of the same type under a single variable name.

In the next chapter, we are going to learn about building classes.

Exercises

1. Write a Java program that takes two numbers as input and prints their sum, difference, product, and quotient.
2. Write a program that takes a number as input and checks whether it is positive, negative, or zero using an if-else statement.
3. Write a Java program that calculates the grade based on a student's percentage using an if-else-if ladder:
    a. 90 and above → "A"
    b. 80 - 89 → "B"
    c. 70 - 79 → "C"
    d. 60 - 69 → "D"
    e. Below 60 → "F"
4. Write a Java program using a switch-case statement to take a day number (1-7) as input and print the corresponding day of the week (1 = Monday, 2 = Tuesday, ..., 7 = Sunday).
5. Write a Java program that prints the first 10 Fibonacci numbers using a loop.
6. Write a Java program that finds the first 5 multiples of 7 using a loop and exits when the number exceeds 50 (use break statement).
7. Write a Java program that skips printing multiples of 5 between 1 and 50 (use continue statement).
8. Write a Java program that takes 5 numbers as input from

the user, stores them in an array, and prints the sum and average.

9. Write a Java program that finds the largest and smallest numbers in an array.
10. Write a Java program that checks if a given number is present in an array (linear search).
11. Write a Java program to initialize and print a 3x3 matrix.
12. Write a Java program that adds two 3x3 matrices.

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 3
Building Classes

Introduction

This chapter of the book covers the fundamentals of creating classes, including properties, methods, and constructors, which form the basis for object-oriented design. It also covers access modifiers, method overloading, and garbage collection. The chapter explains this keyword, static members, and the final keyword. Lastly, it deals with wrapper classes which represent primitive data types as objects, offering utility methods for conversion and manipulation, and the String class and its methods for text processing tasks in Java applications, facilitating string manipulation, comparison, and formatting.

Structure

The chapter covers the following topics:

- Introducing class
- Constructors
- Object access modifiers
- Method overloading

- Garbage collection
- ‘this’ keyword
- ‘static’ keyword
- ‘final’ keyword
- Wrapper classes
- String class and methods

Objectives

The objective of the chapter is to provide a comprehensive understanding of key concepts and features in Java **object-oriented programming** (**OOP**). By the end of this chapter, you will be able to grasp the fundamentals of class design and instantiation. Understanding object access modifiers enables learners to control the visibility and accessibility of class members, ensuring proper encapsulation and information hiding. You will also learn method overloading and garbage collection mechanisms. Learners will get complete clarity on different use cases of 'this' keyword, static members, and 'final' keyword. Delving into wrapper classes enables the manipulation of primitive data types as objects, facilitating interoperability between primitive types and objects. Lastly, exploring the String class and its methods equips learners with powerful tools for string manipulation, searching, and modification.

Introducing class

We discussed the class in previous chapters. Although, the program structure we learnt in previous chapters was basic implementation of **main()** method to introduce Java syntax. Let us explore the real power of class.

Class can be defined as a template/blueprint that describes the behaviors/states of a particular entity. It is the core of Java language. A class defines a new data type. Once defined, this new type can be used to create the object of that type. It is also known as a **user-defined datatype**.

Object is an instance of class. You may call it the physical existence of a logical template class. A class contains both data (referred to as

fields or variables) and methods (functions or behaviors) that define the characteristics and behaviors of objects instantiated from it. The data or variables defined within a class are called instance variables. The instance variables and methods are known as **class members**.

A class and an object can be related as follows: Consider an ice tray (like a cube shape) as a class. Then, ice cubes can be considered as the objects of the ice tray.

A class is declared using **class** keyword. A general form of class is shown below:

```
class className{
datatype instance-variable1;
datatype instance-variable2;
datatype instance-variableN;
datatype method_name1(parameter_list)
{
//Method body
}
datatype method_name2(parameter_list)
{
//Method body
}
datatype method_nameN(parameter_list)
{
//Method body
}
}
```

The fields or variables of a class are referred to as properties of a class, which represent the attributes or data associated with objects created from that class. The variable declared inside a class is called an instance variable because each instance of the class (i.e., each object) contains its own copy of these variables. Methods in a class are essentially functions that define the behavior or actions that objects of that class can perform. They encapsulate the logic and functionality related to the class, allowing objects to interact with each other and with external systems.

Simple class example

Suppose, **Student** is a class and student's roll number, and three subject marks are its properties.

Let us see this in the Java syntax:

```
1. class Student
2. {
3. int rollno;
4. int marks1;
5. int marks2;
6. int marks3;
7. }
```

This declaration creates a new data type with name **Student**. Now, this name can be used to declare objects of type **Student**. Declaration of a class creates a template only; no actual object has been created so far. To create an object of type student the statement should be:

Student myStudent = new Student(); //creates a Student object myStudent.

The following figure illustrates the creation of an object and its memory mapping:

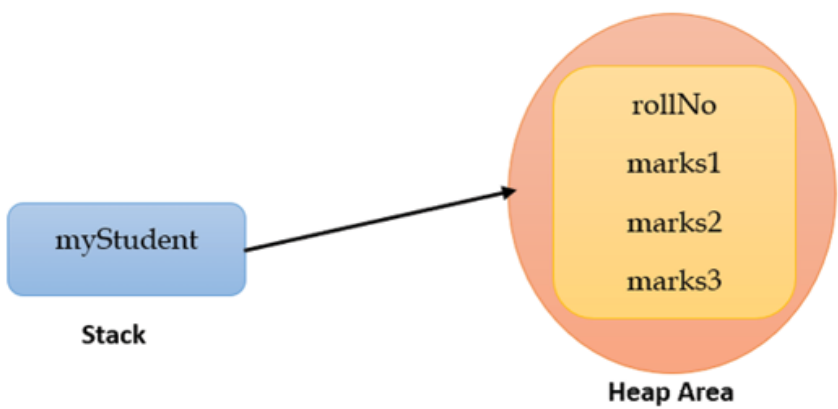


***Figure 3.1**: Object declaration*

Object **myStudent** is the physical existence of **Student** class. **new** operator dynamically allocates memory for an object and returns a reference to it. Each object of the class will have its own copy of instance variables **rollno, marks1, marks2**, and **marks3**. To access these variables **dot(.)** operator is used. For example, to assign **rollno**

of **myStudent** as **1**, the statement will be:

myStudent.rollno = 1;

The dot operator is used to access instance variables and methods within an object. This is also called a **membership operator**. Let us see a complete program using student class:

```
1. /* A program that uses the Student class
2. call this file StudentDemo.java */
3. class Student
4. {
5. int rollno;
6. int marks1;
7. int marks2;
8. int marks3;
9. }
10. //This class declares an object of type Student
11. class StudentDemo
12. {
13. public static void main(String arg[])
14. {
15. int total;
16. Student std1 = new Student();
17. //assign values to std1's instance variables
18. std1.rollno = 1;
19. std1.marks1 = 56;
20. std1.marks2 = 74;
21. std1.marks3 = 65;
22. //compute total marks of student
23. total = std1.marks1 + std1.marks2 + std1.marks3;
24. System.out.println(" Total marks obtained :" + total);
25. }
26. }
```

The file containing this program should be named **StudentDemo.java**, because the **StudentDemo** class contains the **main** method. Compilation of this program will create two classes, **Student** and **StudentDemo**. To run this program, execute the **StudentDemo** class. This will give the following output:

Total marks obtained :195

As stated earlier, each object has its own copy of instance variables.

Therefore, in the above program, if two objects of **Student** class are created, both will have their own separate copies of instance variables. Changes made in one object's instance variable will not affect the instance variable of another object.

Adding methods to class

In the above program, we saw a class with instance variables, also called as **properties of the class**. To access the properties, the class provides methods which implement the behavior of the class. Methods provide an interface to class by allowing to hide the actual implementation details behind the cleaner method abstraction. Methods can be defined to access the data as well as for implementing the application logic.

Let us add a method to our **Student** class:

```
1. /* A program that uses the Student class
2. call this file StudentDemo.java */
3. class Student
4. {
5. int rollno;
6. int marks1;
7. int marks2;
8. int marks3;
9. //Display total marks of a Student
10. void totalMarks()
11. {
12. System.out.println(" Total marks obtained by Roll no :" + rollno);
13. System.out.println(marks1 + marks2 + marks3);
14. }
15. }
16. //This class declares an object of type Student
17. class StudentDemo
18. {
19. public static void main(String arg[])
20. {
21. Student std1 = new Student();
22. Student std2 = new Student();
23.
24. //assign values to std1's instance variables
25. std1.rollno = 1;
```

```
26. std1.marks1 = 56;
27. std1.marks2 = 74;
28. std1.marks3 = 65;
29.
30. //assign values to std2's instance variables
31. std2.rollno = 2;
32. std2.marks1 = 66;
33. std2.marks2 = 54;
34. std2.marks3 = 57;
35. //display total marks of student1
36. std1.totalMarks();
37.
38. //display total marks of student2
39. std2.totalMarks();
40.
41. }
42. }
```

This program will give the following output:

Total marks obtained by Roll no :1
195
Total marks obtained by Roll no :2
177

Let us analyze the following two lines of code:

```
1. std1.totalMarks();
2. std2.totalMarks();
```

The first line invokes the method **totalMarks()** on object **std1** using the object name **std1** followed by the dot operator. Thus, the first line displays the total marks of **std1**, and the second line displays the total marks of **std2**.

A very important point to note here is in the method **totalMarks()**. The instance variables in the method are referred to directly without any object name. This is because a method is always invoked relative to an object of the class. So, the instance variables of the object, which invoked the method, will be implicitly referred.

Returning value from a method

The previous implementation of **totalMarks()** method displays the computed total marks in the method itself that is within the class

where it belongs. Sometimes, we may need the value in other parts of our program. In that case, the method should return the value to the caller. The following code changed the **totalMarks()** implementation to return the value:

```
1. /* A program that uses the Student class
2. call this file StudentDemo.java */
3. class Student
4. {
5. int rollno;
6. int marks1;
7. int marks2;
8. int marks3;
9. //Display total marks of a Student
10. int totalMarks()
11. {
12. return marks1 + marks2 + marks3;
13. }
14. }
15. //This class declares an object of type Student
16. class StudentDemo
17. {
18. public static void main(String arg[])
19. {
20. Student std1 = new Student();
21. Student std2 = new Student();
22. int total;
23. //assign values to std1's instance variables
24. std1.rollno = 1;
25. std1.marks1 = 56;
26. std1.marks2 = 74;
27. std1.marks3 = 65;
28.
29. //assign values to std2's instance variables
30. std2.rollno = 2;
31. std2.marks1 = 66;
32. std2.marks2 = 54;
33. std2.marks3 = 57;
34. //display total marks of student1
35. total = std1.totalMarks();
36. System.out.println(" Total marks obtained by Roll no 1:" + total);
```

```
37.
38. //display total marks of student2
39. total=std2.totalMarks();
40. System.out.println(" Total marks obtained by Roll no 2:"+total);
41. }
42. }
```

Observe the statement:

total=std1.totalMarks();

The **totalMarks()** method is now returning int value (unlike void in the previous version). Therefore the **main()** method, which is calling **totalMarks()** method will get an integer value, which is stored in a variable named total and then printed by **System.out.println()** method.

The following two statements can be merged together:

```
1. total=std1.totalMarks();
2. System.out.println(" Total marks obtained by Roll no 1:"+total);
```

as:

```
System.out.println(" Total marks obtained by Roll no 1:"+std1.totalMarks());
```

Method with parameters

A method without parameters does not accept any input values when it is called. It performs its task using fixed values or class members. We can write a method with parameters to accept input values, allowing it to perform operations on different data provided at runtime.

For example, the following program shows a method add without a parameter:

```
1. public class Calculator {
2.
3.     // Method with parameters
4.     public int add() {
5.         int sum = 5 + 2;
6.         return sum;
7.     }
8.
9.     public static void main(String[] args) {
10.        // Create an instance of Calculator
```

```
        Calculator calculator = new Calculator();

        // Call the method with parameters
        int result = calculator.add();

        // Print the result
        System.out.println("The sum is: " + result);
    }
}
```

Every time your program execution will give the following output:

The sum is: 7

However, we may want to calculate the sum of different values. Let us change the program:

```
public class Calculator {

    // Method with parameters
    public int add(int num1, int num2) {
        int sum = num1 + num2;
        return sum;
    }

    public static void main(String[] args) {
        // Create an instance of Calculator
        Calculator calculator = new Calculator();

        // Call the method with parameters
        int result = calculator.add(5, 3);

        // Print the result
        System.out.println("The sum is: " + result);
    }
}
```

Now, the method can process different input values each time it is called, making it more flexible and reusable.

Let us enhance the **Student** class by adding a method with parameters to set the values of the instance variables. This makes the code cleaner and demonstrates how parameters can be used to pass information into methods.

```
class Student {
    int rollno;
    int marks1;
    int marks2;
    int marks3;

    // Method to set student details (with parameters)
    void setStudentDetails(int rollno, int marks1, int marks2, int marks3)
        this.rollno = rollno;
        this.marks1 = marks1;
        this.marks2 = marks2;
        this.marks3 = marks3;
    }

    // Display total marks of a Student
    int totalMarks() {
        return marks1 + marks2 + marks3;
    }
}
```

We will update the **StudentDemo** class to use the new **setStudentDetails** method:

```
class StudentDemo {
    public static void main(String arg[]) {
        Student std1 = new Student();
        Student std2 = new Student();
        int total;

        // Assign values to std1's instance variables using method with par
        std1.setStudentDetails(1, 56, 74, 65);

        // Assign values to std2's instance variables using method with par
        std2.setStudentDetails(2, 66, 54, 57);

        // Display total marks of student1
        total = std1.totalMarks();
        System.out.println("Total marks obtained by Roll no 1: " + total);

        // Display total marks of student2
        total = std2.totalMarks();
```

```
        System.out.println("Total marks obtained by Roll no 2: " + total);
    }
}
```

setStudentDetails(int rollno, int marks1, int marks2, int marks3) method in the **Student** class is an example of a method with parameters. It takes four parameters and sets the instance variables of the **Student** objects accordingly. **this** keyword is used to refer to the current object's instance variables to avoid confusion with the method parameters.

In the **StudentDemo** class, we create two **Student** objects: **std1** and **std2**. Instead of directly accessing and setting the instance variables, we use the **setStudentDetails** method to set their values. This encapsulates the data assignment process within the method, making the code more maintainable and readable. The **totalMarks** method is then called on each **Student** object to calculate and print their total marks.

By using a method to set values, we **encapsulate** the logic for setting instance variables, which promotes better coding practices. The **setStudentDetails** method can be reused whenever we need to set the details of a **Student** object. Using methods with parameters makes the code more readable and easier to understand.

Constructors

In the previous example, we manually assigned values to the instance variables using a method with parameters. Java provides a special method called constructor, to achieve the same result more elegantly.

A constructor is a special method that is called when an object is instantiated. It is used to initialize the object's state. Unlike regular methods, constructors do not have a return type, not even void, and their name must exactly match the class name.

Let us add a **Constructor** to the **Student** class:

```
class Student {
    int rollno;
    int marks1;
    int marks2;
    int marks3;
```

```
6.
7.     // Constructor  to initialize the instance variables
8.     public Student() {
9.         this.rollno = 1;
10.        this.marks1 = 55;
11.        this.marks2 = 65;
12.        this.marks3 = 67;
13.    }
14.
15.    // Method to calculate total marks
16.    int totalMarks() {
17.        return marks1 + marks2 + marks3;
18.    }
19. }
```

In order to instantiate an object of the class, **new** keyword is used, followed by the constructor method of the class, which has to be instantiated. For the above-listed class named **Student** with the constructor method **Student ()**, to instantiate the object of the **Student** class and to allocate memory for this class, the expression new **Student()** can be used, which returns a reference to the newly created **Student** object in the memory.

We will update the **StudentDemo** class to use the constructor for initializing **Student** objects:

```
1. class StudentDemo {
2.     public static void main(String[] args) {
3.         // Create Student objects using the constructor
4.         Student std1 = new Student();
5.         int total ;
6.
7.         // Display total marks of student1
8.         total = std1.totalMarks();
9.         System.out.println("Total marks obtained by Roll no 1: " + total);
10.  }
11. }
```

The constructor **public Student()** is defined to initialize the instance variables of the **Student** class. In the **StudentDemo** class, **Student** objects are now created using the constructor with the following syntax:

Student std1 = new Student();

This approach ensures that all instance variables are initialized at the time of object creation, making the code more concise and less error prone.

The **totalMarks** method remains unchanged and is used to calculate the total marks for **Student** object. However, the above-mentioned implementation will assign the same values to all the objects. We can modify the constructor to accept parameters to initialize the values as desired. Let us modify the constructor:

```
1. class Student {
2.     int rollno;
3.     int marks1;
4.     int marks2;
5.     int marks3;
6.
7.     // Constructor with parameters to initialize the instance variables
8.     public Student(int rollno, int marks1, int marks2, int marks3) {
9.         this.rollno = rollno;
10.        this.marks1 = marks1;
11.        this.marks2 = marks2;
12.        this.marks3 = marks3;
13.    }
14.
15.    // Method to calculate total marks
16.    int totalMarks() {
17.        return marks1 + marks2 + marks3;
18.    }
19. }
```

The **StudentDemo** class will be modified to pass the values to the constructor as follows:

```
1. class StudentDemo {
2.     public static void main(String[] args) {
3.         // Create Student objects using the constructor
4.         Student std1 = new Student(1, 56, 74, 65);
5.         Student std2 = new Student(2, 66, 54, 57);
6.         int total;
7.
8.         // Display total marks of student1
```

```
        total = std1.totalMarks();
        System.out.println("Total marks obtained by Roll no 1: " + total);

        // Display total marks of student2
        total = std2.totalMarks();
        System.out.println("Total marks obtained by Roll no 2: " + total);
    }
}
```

Like methods, constructors also vary in the types of parameters they accept.

Benefits of using a constructor

The following are the benefits of using a constructor:

- **Automatic initialization**: Constructors ensure that objects are properly initialized when they are created, reducing the likelihood of errors due to uninitialized variables.
- **Code conciseness**: Initialization code is centralized in the constructor, making the main code more concise and easier to read.
- **Object-oriented design**: Constructors promote a more object-oriented design by encapsulating the initialization logic within the class itself.

Comparison to methods

A constructor in Java is basically a method with some exclusive features. Let us compare the constructor with regular methods:

Constructors	Methods
Constructors can be named with any valid identifier.	Methods can be named with any valid identifier.
Constructors are called explicitly by the user.	Methods are called explicitly by the user.
Constructors do not have a return type (not even void).	Methods have a return type (or void if they do not return a value).
Constructors are used specifically for initializing new objects.	Methods are used to perform various operations.

***Table 3.1**: Comparison between constructor and method*

Object access modifiers

Object access modifiers in Java play a crucial role in encapsulation, a core concept of **object-oriented programming** (**OOP**). They

control the visibility of classes, methods, and variables, thereby defining how the components of a class can be accessed from other parts of the program. The access level or visibility of classes, methods, and variables is defined by access modifiers. They control where a particular class, method, or variable can be accessed from. The four main access modifiers in Java are private, default (no modifier), protected, and public.

Private access modifier

The private access modifier is the most restrictive access level. When a member is declared private, it is accessible only within the class in which it is declared. **private** helps encapsulate an object's internal state, meaning that the internal workings of a class are hidden from the outside world. This allows for better control over how the data is accessed and modified. It prevents unauthorized or unintended access to the internal components of a class, ensuring that the data can only be accessed or modified through well-defined methods.

```
1. public class Example {
2.     private int privateVariable;
3.
4.     private void privateMethod() {
5.         System.out.println("Private method");
6.     }
7. }
```

Access: Access of private members are only within the same class.

Default access modifier

When no access modifier is specified, the member has default (also known as **package-private**) access. This means it is accessible only within the same package.

```
1. class Example {
2.     int defaultVariable;
3.
4.     void defaultMethod() {
5.         System.out.println("Default method");
6.     }
7. }
```

Usage: Default access modifier is used when you want to allow access to classes within the same package but restrict access from outside the package.

Access: Default members are accessed only within the same package.

Protected access modifier

The protected access modifier allows the member to be accessible within the same package and by subclasses (even if they are in different packages).

```
1. public class Example {
2.     protected int protectedVariable;
3. 
4.     protected void protectedMethod() {
5.         System.out.println("Protected method");
6.     }
7. }
```

Usage: Protected access modifier is used when you want to allow access to subclasses and classes within the same package.

Access: Protected members are available within the same package and by subclasses in other packages.

Public access modifier

The public access modifier allows the member to be accessible from any other class:

```
1. public class Example {
2.     public int publicVariable;
3. 
4.     public void publicMethod() {
5.         System.out.println("Public method");
6.     }
7. }
```

Usage: Public access modifier is used when you want to make the member accessible from any other class.

Access: public members are available from any other class in any package.

The following is the comparison table of availability of different access modifiers at different levels:

***Table 3.2**: Comparison table for different Access modifiers*

Method overloading

Method overloading is a feature in Java that allows a class to have more than one method with the same name, but different parameters. These differences can be in the number of parameters, the type of parameters, or both. Method overloading is a compile-time polymorphism feature, meaning the decision about which method to call is made at compile time.

All overloaded methods must have the same name. Overloaded methods must differ in the type, number, or both of their parameters. The return type can be different for overloaded methods, but it alone is not sufficient for method overloading. Method overloading is resolved at compile time based on the method signature (name + parameter list).

Benefits of method overloading

The following are the benefits of method overloading:

- **Code readability**: Having multiple methods with the same name makes the code easier to read and understand.
- **Code maintenance**: It simplifies maintenance because the same logical method can perform different functions based on input parameters.
- **Flexibility**: It provides flexibility to handle different types of input parameters within the same method name.

Let us consider a basic example of method overloading:

```
1. public class Calculator {
2.
3.     // Method to add two integers
```

```
    public int add(int a, int b) {
        return a + b;
    }

    // Method to add three integers
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method to add two double values
    public double add(double a, double b) {
        return a + b;
    }

    public static void main(String[] args) {
        Calculator calc = new Calculator();

        System.out.println("Sum of two integers: " + calc.add(5, 10)); // C
        System.out.println("Sum of three integers: " + calc.add(5, 10, 15)
        System.out.println("Sum of two doubles: " + calc.add(5.5, 10.5));
    }
}
```

In this example, the **add** method is overloaded three times with different parameter lists. The correct method is chosen based on the arguments passed during the method call. Let us consider another example of overloading with different parameter types:

```
public class Printer {

    // Method to print a string
    public void print(String s) {
        System.out.println("String: " + s);
    }

    // Method to print an integer
    public void print(int i) {
        System.out.println("Integer: " + i);
    }

    // Method to print a double
```

```
    public void print(double d) {
        System.out.println("Double: " + d);
    }

    public static void main(String[] args) {
        Printer printer = new Printer();

        printer.print("Hello, World!"); // Calls print(String)
        printer.print(123); // Calls print(int)
        printer.print(12.34); // Calls print(double)
    }
}
```

In this example, the print method is overloaded to handle different types of arguments (String, int, and double). The appropriate method is called based on the type of argument passed.

The method can also be overloaded with a varying number of parameters. See the following example:

```
public class AreaCalculator {

    // Method to calculate the area of a rectangle
    public double area(double length, double width) {
        return length * width;
    }

    // Method to calculate the area of a square
    public double area(double side) {
        return side * side;
    }

    public static void main(String[] args) {
        AreaCalculator calc = new AreaCalculator();

        System.out.println("Area of rectangle: " + calc.area(5.0, 10.0)); //
        System.out.println("Area of square: " + calc.area(5.0)); // Calls are
    }
}
```

In this example, the area method is overloaded with different numbers of parameters. One method calculates the area of a rectangle, while the other calculates the area of a square.

Garbage collection

A crucial aspect of memory management is handling the memory used by unreferenced objects. **Garbage collection** (**GC**) in Java is an automatic memory management process that helps in reclaiming memory occupied by objects that are no longer in use by the program. Java applications use a portion of memory called the heap to store objects created during the program's execution. When objects are no longer needed, they become eligible for garbage collection. This helps in optimizing memory usage and preventing memory leaks. Memory leaks can lead to application performance degradation or crashes. The garbage collector performs reachability analysis. Objects that are not reachable are deemed garbage and eligible for collection. The memory occupied by such objects can be reclaimed. Garbage collection occurs primarily when the JVM cannot allocate memory for new objects because the heap is full, prompting a collection to reclaim space. It also takes place at regular intervals based on the JVM's internal heuristics to optimize memory usage and prevent fragmentation. Additionally, developers can request garbage collection explicitly using **System.gc()**, though this is merely a suggestion to the JVM. Different Java runtime implementation takes varying approach to garbage collection. The purpose is to manage memory automatically, allowing developers to focus more on application logic rather than memory management.

‘this’ keyword

The **this** keyword is a reference variable that refers to the current object. It is used within an instance method or constructor to access members of the current object. The **this** keyword helps in differentiating between instance variables and local variables or parameters, as well as invoking other constructors in the same class.

Uses of ‘this’ keyword

In this section, we will discuss the uses of **this** keyword:

- **To refer current class instance variables**: When there is a naming conflict between instance variables and parameters or local variables, this is used to differentiate them.

```
1. public class Student {
```

```
    private String name;
    private int age;

    public Student(String name, int age) {
        this.name = name; // 'this.name' refers to the instance variabl
        this.age = age;   // 'this.age' refers to the instance variable, 'a
    }

    public void display() {
        System.out.println("Name: " + this.name + ", Age: " + this.a
    }

    public static void main(String[] args) {
        Student student = new Student("John", 20);
        student.display(); // Output: Name: John, Age: 20
    }
}
```

- **To invoke current class methods**: You can use this to call another method of the current class. Although it is not mandatory, it can be used for clarity.

```
public class Calculation {
    int a, b;

    public void setValues(int a, int b) {
        this.a = a;
        this.b = b;
    }

    public void displayValues() {
        System.out.println("a: " + a + ", b: " + b);
    }

    public void performTask() {
        this.setValues(10, 20); // Invoking method using 'this'
        this.displayValues();   // Invoking method using 'this'
    }

    public static void main(String[] args) {
        Calculation calc = new Calculation();
```

```
        calc.performTask(); // Output: a: 10, b: 20
    }
}
```

- **To invoke current class constructor**: This can be used to call one constructor from another constructor in the same class. This is known as **constructor chaining**.

```
public class Box {
    private int width, height, depth;

    public Box() {
        this(10, 10, 10); // Invoking parameterized constructor
    }

    public Box(int width, int height, int depth) {
        this.width = width;
        this.height = height;
        this.depth = depth;
    }

    public void display() {
        System.out.println("Dimensions: " + width + "x" + height +
    }

    public static void main(String[] args) {
        Box box = new Box();
        box.display(); // Output: Dimensions: 10x10x10
    }
}
```

- **To return current class instance**: A method can return the current class instance using this.

```
public class Builder {
    private String field;

    public Builder setField(String field) {
        this.field = field;
        return this; // Returning current instance
    }

    public void display() {
        System.out.println("Field: " + field);
```

```
    }

    public static void main(String[] args) {
        Builder builder = new Builder();
        builder.setField("Value").display(); // Method chaining
    }
}
```

- **To pass as an argument in method call**: This can be passed as an argument in the method call.

```
public class Data {
    int value;

    public Data(int value) {
        this.value = value;
    }

    public void display(Data obj) {
        System.out.println("Value: " + obj.value);
    }

    public void show() {
        display(this); // Passing 'this' as an argument
    }

    public static void main(String[] args) {
        Data data = new Data(50);
        data.show(); // Output: Value: 50
    }
}
```

- **To pass as an argument in constructor call**: This can be passed as an argument in the constructor call.

```
class A {
    B b;

    A() {
        b = new B(this); // Passing 'this' as an argument in constructo
    }

    public void display() {
```

```
        System.out.println("Inside Class A");
    }
}

class B {
    A a;

    B(A a) {
        this.a = a;
        a.display();
    }
}

public class Test {
    public static void main(String[] args) {
        A a = new A(); // Output: Inside Class A
    }
}
```

‘static’ keyword

static keyword is used to create class variable and class methods which can be accessed without an instance of a class. The **static** keyword is used to indicate that a particular member belongs to the class itself rather than instances of the class. This means that static members are shared among all instances of the class. The **static** keyword can be applied to variables, methods, blocks, and nested classes. In the following sections, we will see a detailed explanation of each.

‘static’ variables

static variables (also known as **class variables**) are variables declared with the **static** keyword. They are shared among all instances of the class, meaning there is only one copy of the variable in memory, regardless of how many objects are created. **static** variables are initialized only once. **static** variable is used to represent the common property of a class.

```
public class Counter {
    static int count = 0; // Static variable
```

```
3.
4.    public Counter() {
5.        count++;
6.    }
7.
8.    public static void main(String[] args) {
9.        Counter c1 = new Counter();
10.       Counter c2 = new Counter();
11.       Counter c3 = new Counter();
12.
13.       System.out.println("Number of instances created: " + Counter.cou
14.   }
15. }
```

'static' methods

static methods are methods declared with the **static** keyword. They belong to the class rather than any instance, and therefore, they can be called without creating an instance of the class. **static** methods can access **static** variables and other static methods directly. These methods cannot access instance variables or instance methods directly. Often used for utility or helper methods. **static** method can be accessed using the class name.

```
1. public class MathUtils {
2.    public static int add(int a, int b) {
3.        return a + b;
4.    }
5.
6.    public static void main(String[] args) {
7.        int sum = MathUtils.add(5, 10); // No need to create an instance
8.        System.out.println("Sum: " + sum); // Output: Sum: 15
9.    }
10. }
```

'static' blocks

static blocks are blocks of code that are executed when the class is loaded into memory. They are used to initialize static variables or perform other static initialization tasks. Static blocks are executed in the order they appear in the class. Executed when the class is loaded. Static block can be used to initialize static variables. A class

can have multiple static blocks:

```
public class StaticBlockExample {
    static int data;

    static {
        data = 50; // Static block to initialize static variable
        System.out.println("Static block executed. Data initialized to " + d
    }

    public static void main(String[] args) {
        System.out.println("Data: " + StaticBlockExample.data); // Output:
    }
}
```

Let us discuss a detailed example integrating all static features:

```
public class StaticExample {
    static int staticVariable;
    int instanceVariable;

    // Static block to initialize static variable
    static {
        staticVariable = 10;
        System.out.println("Static block executed.");
    }

    // Static method
    public static void displayStatic() {
        System.out.println("Static Variable: " + staticVariable);
        // Cannot access instanceVariable here
    }

    // Instance method
    public void displayInstance() {
        System.out.println("Instance Variable: " + instanceVariable);
        System.out.println("Static Variable: " + staticVariable); // Can acc
    }

    public static void main(String[] args) {
        // Accessing static method without creating an instance
        StaticExample.displayStatic();
```

```
26.
27.         // Creating instances
28.         StaticExample obj1 = new StaticExample();
29.         obj1.instanceVariable = 20;
30.         obj1.displayInstance();
31.
32.         StaticExample obj2 = new StaticExample();
33.         obj2.instanceVariable = 30;
34.         obj2.displayInstance();
35.
36.         // Accessing static variable using class name
37.         System.out.println("Static Variable accessed via class name: " + St
38.     }
39. }
```

The program will give the following output:

Static block executed.
Static Variable: 10
Instance Variable: 20
Static Variable: 10
Instance Variable: 30
Static Variable: 10
Static Variable accessed via class name: 10

In this example, the static block initializes the **staticVariable** when the class is loaded. The **staticVariable** is accessed both within static and instance methods. The **staticVariable** retains its value across different instances of the class. The **displayStatic** method demonstrates how static methods can be called without creating an instance of the class.

‘final’ keyword

The **final** keyword is used to apply restrictions on classes, methods, and variables. When applied, it ensures that the entity it modifies cannot be changed or extended. Let us explore how final is used in different contexts:

‘final’ variables

A **final** variable can be a constant. Once initialized, its value cannot be changed. A final variable must be initialized when it is declared

or in the constructor. The value of a final variable cannot be modified once it is set. Final variables can be used to define constants.

Example:

```
1. public class Constants {
2.     public static final double PI = 3.14159; // Final static variable
3.     public final int MAX_AGE;
4.
5.     public Constants(int maxAge) {
6.         this.MAX_AGE = maxAge; // Final instance variable must be initial
7.     }
8.
9.     public static void main(String[] args) {
10.        Constants constants = new Constants(100);
11.        System.out.println("PI: " + PI);
12.        System.out.println("MAX_AGE: " + constants.MAX_AGE);
13.        // constants.MAX_AGE = 120; // This would cause a compilation er
14.    }
15. }
```

'final' methods

A **final** method cannot be overridden by subclasses. This is useful for ensuring that the behavior of a method remains consistent across all subclasses.

Example:

```
1. class Parent {
2.     public final void show() {
3.         System.out.println("This is a final method.");
4.     }
5. }
6.
7. class Child extends Parent {
8.     // public void show() { // This would cause a compilation error
9.     //     System.out.println("Cannot override final method.");
10.    // }
11. }
12.
13. public class FinalMethodExample {
```

```
    public static void main(String[] args) {
        Child child = new Child();
        child.show(); // Output: This is a final method.
    }
}
```

'final' classes

A **final** class cannot be subclassed. This is useful for creating immutable classes or classes that should not be extended for security or design reasons. The **final** keyword when used in class declaration, prevents inheritance and ensures the class's implementation cannot be altered by subclasses.

Example:

```
public final class ImmutableClass {
    private final String name;

    public ImmutableClass(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

// class SubClass extends ImmutableClass { // This would cause a comp
// }

public class FinalClassExample {
    public static void main(String[] args) {
        ImmutableClass obj = new ImmutableClass("Immutable");
        System.out.println("Name: " + obj.getName()); // Output: Name: I
    }
}
```

Wrapper classes

Wrapper classes in Java are a part of the **java.lang** package and are used to convert primitive data types into objects. Each primitive

data type (such as int, boolean, char, etc.) has a corresponding wrapper class (such as Integer, Boolean, Character, etc.). These classes are useful because they allow primitives to be used in places where only objects are allowed, such as in collections (like ArrayList, HashMap, etc.) and in object-oriented programming constructs.

Here is an overview of the primitive types and their corresponding wrapper classes:

byte -> Byte
short -> Short
int -> Integer
long -> Long
float -> Float
double -> Double
char -> Character
boolean -> Boolean

The process of converting primitive data types to their corresponding wrapper class and vice versa is called **autoboxing** and **unboxing**. Autoboxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an int to an Integer. Unboxing is the reverse process, where the wrapper class is converted back to its corresponding primitive type.

The following example shows the process:

```
int primitiveInt = 5;
Integer wrapperInt = primitiveInt; // Autoboxing
int anotherPrimitiveInt = wrapperInt; // Unboxing
```

Wrapper classes provide various utility methods for converting between different types, parsing strings into primitives, and more. For example:

- **Integer.parseInt(String s): converts a string to an int.**
- **Double.toString(double d): converts a double to a String.**
- **Character.isDigit(char ch): checks if a character is a digit.**

All wrapper classes are immutable, meaning that once an object is created, it cannot be changed. For example, once an Integer object is created with a value, that value cannot be modified. Wrapper

classes like Integer, Short, Byte, Character, and Long cache objects for certain ranges of values. For example, Integer caches objects for values between -128 and 127. This means that for these values, the same object reference is returned, which can improve performance.

Here is a practical example demonstrating the use of wrapper classes:

```
1. import java.util.ArrayList;
2.
3. public class WrapperClassExample {
4.     public static void main(String[] args) {
5.         // Using wrapper classes with collections
6.         ArrayList<Integer> numbers = new ArrayList<>();
7.
8.         // Autoboxing: converting int to Integer
9.         numbers.add(10);
10.        numbers.add(20);
11.        numbers.add(30);
12.
13.        // Unboxing: converting Integer to int
14.        int sum = 0;
15.        for (Integer number : numbers) {
16.            sum += number;  // unboxing
17.        }
18.
19.        System.out.println("Sum: " + sum);
20.
21.        // Using utility methods
22.        int maxInt = Integer.MAX_VALUE;
23.        System.out.println("Max Integer Value: " + maxInt);
24.
25.        boolean isDigit = Character.isDigit('5');
26.        System.out.println("'5' is a digit: " + isDigit);
27.    }
28. }
```

In this example, **Integer** objects are used in an **ArrayList**. Autoboxing and unboxing are demonstrated when adding int values to the list and summing them up. Utility methods from the **Integer** and **Character** classes are used to showcase some of the functionality provided by wrapper classes.

String class and methods

The String class in Java is a part of the **java.lang** package and represents a sequence of characters. Strings are immutable in Java, meaning once a String object is created, it cannot be changed. The String class provides a wide range of methods for manipulating and handling strings.

Strings can be created in several ways:

- **String literals**:

  String str1 = "Hello, World!";

- **Using the new keyword**:

  String str2 = new String("Hello, World!");

Here are some of the most commonly used methods in the **String** class:

- **Length**:

  The **length()** method is used to determine the number of characters (including spaces) in a string. It returns the length of the string as an integer:

  int length()

  Returns the number of characters in the string:

  int len = str1.length();

- **Character access**:

  char charAt(int index)

  Returns the character at the specified index:

  char ch = str1.charAt(0); // 'H'

- **Substring**:

  String substring(int beginIndex)

  Returns a new string that is a substring starting from the specified index to the end of the string:

  String substring(int beginIndex, int endIndex)

  Returns a new string that is a substring from the specified beginIndex to endIndex – 1:

  String sub1 = str1.substring(7); // "World!"
  String sub2 = str1.substring(7, 12); // "World"

- **Concatenation**:

  String concat(String str)

  Concatenates the specified string to the end of this string:

  String str3 = str1.concat(" How are you?");

- **Contains**:

  boolean contains(CharSequence s)

  Returns true if the string contains the specified sequence of characters:

  boolean containsWorld = str1.contains("World"); // true

- **Index of**:

  int indexOf(int ch)

  Returns the index of the first occurrence of the specified character:

  int indexOf(String str)

  Returns the index of the first occurrence of the specified substring:

  int index = str1.indexOf('W'); // 7
  int index2 = str1.indexOf("World"); // 7

- **Replace**:

  String replace(char oldChar, char newChar)

  Replaces all occurrences of the specified character with the new character:

  String replace(CharSequence target, CharSequence replacement)

  Replaces each substring that matches the specified literal target sequence with the specified literal replacement sequence:

  String str4 = str1.replace('o', '0'); // "Hell0, W0rld!"
  String str5 = str1.replace("World", "Java"); // "Hello, Java!"

- **Trim**:

  String trim()

  Returns a copy of the string with leading and trailing whitespace removed:

String str6 = " Hello, World! ";
String trimmed = str6.trim(); // "Hello, World!"

- **To lower case / to upper case:**

  String toLowerCase()

  Converts all characters in the string to lower case:

  String toUpperCase()

  Converts all characters in the string to upper case:

  String lower = str1.toLowerCase(); // "hello, world!"
  String upper = str1.toUpperCase(); // "HELLO, WORLD!"

- **Split**:

  String[] split(String regex)

  Splits the string around matches of the given regular expression.

  String[] words = str1.split(", "); // ["Hello", "World!"]

- **Value of**:

  static String valueOf(int i)

  Returns the string representation of the specified primitive type:

  String numberStr = String.valueOf(123); // "123"

Here is a practical example demonstrating some of the methods:

```
1. public class StringExample {
2.     public static void main(String[] args) {
3.         String greeting = "Hello, World!";
4.
5.         // Length
6.         System.out.println("Length: " + greeting.length());
7.
8.         // Character at
9.         System.out.println("Character at index 1: " + greeting.charAt(1));
10.
11.        // Substring
12.        System.out.println("Substring from index 7: " + greeting.substring
13.        System.out.println("Substring from index 7 to 12: " + greeting.sub
14.
15.        // Concatenation
16.        String extendedGreeting = greeting.concat(" How are you?");
```

```
        System.out.println("Concatenated String: " + extendedGreeting);

        // Contains
        System.out.println("Contains 'World': " + greeting.contains("World

        // Index of
        System.out.println("Index of 'W': " + greeting.indexOf('W'));

        // Replace
        System.out.println("Replace 'World' with 'Java': " + greeting.repla

        // Trim
        String withSpaces = "  Hello, World!  ";
        System.out.println("Trimmed String: '" + withSpaces.trim() + "'");

        // To Lower Case / To Upper Case
        System.out.println("Lower Case: " + greeting.toLowerCase());
        System.out.println("Upper Case: " + greeting.toUpperCase());

        // Split
        String[] parts = greeting.split(", ");
        System.out.println("Split parts: " + parts[0] + " and " + parts[1]);
    }
}
```

This example covers various **String** class methods and demonstrates how they can be used to manipulate and handle strings in Java.

Conclusion

In this chapter, we have learned the basic structure of classes and examined how constructors are used to initialize objects. The chapter then delved into object access modifiers, highlighting the significance of controlling access levels to class members for better encapsulation and security. Method overloading was discussed as a means to enhance the flexibility of methods within classes. We also covered garbage collection and various keywords like this keyword static and final keyword. We also touched upon wrapper classes, which provide a way to use primitive data types as objects and explored the extensive capabilities of the String class and its

methods for handling and manipulating text data.

In the next chapter, we are going to learn about inheritance.

Exercises

1. Create a class Book with instance variables title, author, and price. Add a method displayDetails() that prints the book details. Create multiple Book objects in the main method and call displayDetails().
2. Modify the Book class from the previous exercise to include:
    a. A default constructor that sets default values.
    b. A parameterized constructor that initializes values based on user input.
    c. A method to display book details.
3. Create a class BankAccount with private variables: accountNumber, accountHolder, and balance. Implement getter and setter methods for these variables. Test them in a main method to update and display account details.
4. Write a Java class Calculator that has overloaded methods to perform addition:
    a. add(int a, int b)
    b. add(double a, double b)
    c. add(int a, int b, int c)

    Test these methods in the main method.

1. Create a class Employee with instance variables: id, name, salary. Use the this keyword to initialize values through a constructor. Display the details using a method.
2. Create a class Student with instance variables name and rollNumber and a static variable schoolName. Assign schoolName a common value for all students. Create multiple Student objects and display their details.
3. Create a class Car with a final method showDetails(). Try to override it in a subclass SportsCar and observe the error.
4. Write a Java program that:

a. Accepts a full name as input and extracts the first name using substring().
b. Converts the name into uppercase and lowercase.
c. Finds the length of the name.
d. Checks if the name contains the word "Java".

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 4
Inheriting Behavior

Introduction

Building upon the concept of classes, this chapter explores inheritance, a core principle of object-oriented programming. Readers will understand how classes can inherit properties and behaviors from parent classes, facilitating code reuse and promoting modular design. The chapter also discusses super keyword, which is used to refer to immediate parent class object. While inheriting from a class, the base class method can be redefined in the derived class. This phenomenon is called method overriding. In some scenarios, class declaration is incomplete. It cannot be instantiated directly. Such classes are called abstract classes. An abstract class can be used by inheriting it. We will study all these concepts to further enhance the understanding of inheritance.

Structure

This chapter covers the following topics:

- Introducing inheritance
- Types of inheritance
- ‘super’ keyword

- Method overriding
- Covariant return type
- Role of final keyword in inheritance
- Abstract class

Objectives

The objective of this chapter is to provide a comprehensive understanding of key object-oriented programming concepts related to inheritance in Java. covering various types such as single, multilevel, hierarchical, and hybrid inheritance. Students will learn the use of the super keyword to access superclass methods and constructors, and the concept of method overriding to enable polymorphism. Additionally, the chapter will also cover covariant return types, allowing for more specific return types in overridden methods, and the use of abstract classes to create a common base for subclasses with abstract methods. By the end of the chapter, students will learn to implement these concepts and will be able to develop modular, reusable, and maintainable code.

Introducing inheritance

Inheritance is a fundamental concept in **object-oriented programming** (**OOP**) that allows a new class to inherit the properties and behaviors (fields and methods) of an existing class.

When a class extends another class it inherits all non-private members, including fields and methods. This existing class is known as the superclass (or parent class), and the new class is referred to as the subclass (or child class). Inheritance defines “is-a” relationship between a super class and its sub class. Super class defines common traits for a set of related entities. Sub class is a specialized version of super class. The **extends** and **implements** keywords are used to achieve inheritance in Java. **extends** keyword is used when one class is inheriting from another class or one interface is inheriting from another interface. Implements keyword is used to inherit one interface in a class:

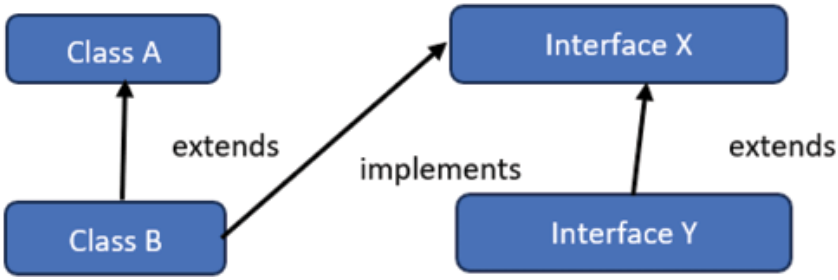


***Figure 4.1**: Demonstration of extends and implements keyword*

Inheritance promotes code reuse and establishes a natural hierarchical relationship between classes, which can simplify the design and implementation of complex systems. With inheritance, subclasses can extend the functionality of super classes by adding new fields and methods or by overriding existing methods to provide specific implementations. This capability is crucial for achieving polymorphism, where a subclass instance can be treated as an instance of its superclass, enabling more flexible and maintainable code.

The general syntax of a class declaration to inherit from a super class is:

class subClassName extends superClassName{
// class implementation
}

To demonstrate the concept, let us see a program:

```
1. // Parent class
2. class Vehicle {
3.     String brand;
4.     int year;
5.
6.     Vehicle(String brand, int year) {
7.         this.brand = brand;
8.         this.year = year;
9.     }
10.
11.    void displayInfo() {
12.        System.out.println("Brand: " + brand + ", Year: " + year);
13.    }
14. }
```

```
15.
16. // Child class
17. class Car extends Vehicle {
18.     int mileage;
19.
20.     Car(String brand, int year, int mileage) {
21.         super(brand, year);
22.         this.mileage = mileage;
23.     }
24.
25.     void displayMileage() {
26.         System.out.println("Mileage: " + mileage + " km");
27.     }
28. }
29.
30. // Main class to test the inheritance
31. public class inheritanceDemo {
32.     public static void main(String[] args) {
33.         // Create an object of the Vehicle class
34.         Vehicle myVehicle = new Vehicle("Toyota", 2010);
35.         myVehicle.displayInfo();
36.
37.         // Create an object of the Car class
38.         Car myCar = new Car("Honda", 2015, 50000);
39.         myCar.displayInfo();
40.         myCar.displayMileage();
41.     }
42. }
```

In this program, **Vehicle** class (Parent class) contains fields brand and year and a constructor to initialize these fields. Method **displayInfo()** prints the brand and year of the vehicle. **Car** class (Child class) inherits from **Vehicle**. This contains an additional field mileage. The constructor of the **Car** class initializes the brand and year using the super keyword and also initializes mileage. It also has a method **displayMileage()** to print the mileage of the car. The third class.

inheritanceDemo is the **main** class which contains the **main** method to run the program. **main** method creates an instance of **Vehicle** and calls its **displayInfo()** method. It also creates an instance of **Car**,

calls its **displayInfo()** method (inherited from **Vehicle**), and calls its **displayMileage()** method.

When you run this program, the output will be:

Brand: Toyota, Year: 2010
Brand: Honda, Year: 2015
Mileage: 50000 km

Although the derived class inherits all members of the base class, it cannot access the members of the base class, which are declared as private. Let us see how different access modifiers work with inheritance. Consider the following example program:

```
1. // Parent class
2. class Vehicle {
3.     public String brand;         // Public member
4.     protected int year;          // Protected member
5.     String model;                // Default (package-
   private) member
6.     private double price;        // Private member
7.
8.     // Constructor
9.     Vehicle(String brand, int year, String model, double price) {
10.        this.brand = brand;
11.        this.year = year;
12.        this.model = model;
13.        this.price = price;
14.    }
15.
16.    // Public method to access private member
17.    public double getPrice() {
18.        return price;
19.    }
20.
21.    // Protected method
22.    protected void displayYear() {
23.        System.out.println("Year: " + year);
24.    }
25.
26.    // Default method
27.    void displayModel() {
28.        System.out.println("Model: " + model);
```

```
    }

    // Private method
    private void displayPrice() {
        System.out.println("Price: " + price);
    }
}

// Child class
class Car extends Vehicle {
    int mileage;

    // Constructor
    Car(String brand, int year, String model, double price, int mileage) {
        super(brand, year, model, price);
        this.mileage = mileage;
    }

    void displayCarInfo() {
        System.out.println("Brand: " + brand);    // Public: accessible
        System.out.println("Year: " + year);      // Protected: accessible
        System.out.println("Model: " + model);    // Default: accessible wi
        // System.out.println("Price: " + price); // Private: not accessible
        System.out.println("Mileage: " + mileage);
    }
}

// Main class to test the access modifiers
public class memberAccessDemo {
    public static void main(String[] args) {
        // Create an object of the Car class
        Car myCar = new Car("Honda", 2015, "Civic", 20000.0, 50000);

        // Accessing members through the child class
        myCar.displayCarInfo();

        // Accessing public method to get private member value
        System.out.println("Price: " + myCar.getPrice());

        // Accessing protected method
```

```
        myCar.displayYear();

        // Accessing default method
        myCar.displayModel();

        // Private method can't be accessed from outside the class
        // myCar.displayPrice(); // This will cause a compilation error
    }
}
```

In this example, brand and **getPrice()** are public members, so can be accessed from both the **Car** class and the **memberAccessDemo** class. Protected members (year and **displayYear()**) are accessible within the same package and subclasses. In this example, year and **displayYear()** can be accessed from the **Car** class because it is a subclass of **Vehicle**. Default members, which are also known as Package-Private (**model** and **displayModel()** in this example), are accessible only within the same package, so they are accessible from **Car**. Private members (**price** and **displayPrice()**) are accessible only within the same class. In this example, price and **displayPrice()** cannot be accessed directly from the **Car** class or any other class outside **Vehicle**. However, the price can be accessed indirectly through the public method **getPrice()**.

When you run the **memberAccessDemo** class, the output will be:
Brand: Honda
Year: 2015
Model: Civic
Mileage: 50000
Price: 20000.0
Year: 2015
Model: Civic

Types of inheritance

Java supports various types of inheritance, including single inheritance (where a subclass inherits from one superclass), multilevel inheritance (where a class is derived from another derived class), hierarchical inheritance (multiple subclasses inherit from a single superclass), and hybrid inheritance (a combination of two or more types, usually implemented through interfaces to avoid

the limitations of multiple inheritance).

Single inheritance

In single inheritance, a subclass inherits from one superclass. See the following example. In this example program, a super class with the name **Animal** is created. This class is inherited by class **Dog,** which is a sub class of class **Animal**. Observe the keyword **extends** is used to inherit the base class **Animal** in the derived class **Dog**.

```
1. class Animal {
2.     void eat() {
3.         System.out.println("This animal eats food.");
4.     }
5. }
6. 
7. class Dog extends Animal {
8.     void bark() {
9.         System.out.println("The dog barks.");
10.    }
11. }
```

The base class **Animal** has one method, **eat()**. Derived class **Dog** adds one more method **bark()**. The objects of the **Dog** class will have access to both the methods, **bark()** from its own class and **eat()** from the base class. In this example, although, **Animal** is super class for **Dog**, it can also be used independently. A sub class can be a super class for another subclass.

Multilevel inheritance

As we saw in the previous section, a subclass can be further extended up to many levels. In multilevel inheritance, a class is derived from another derived class, forming a chain of inheritance. In the following example, the program class **Puppy** extends from **Dog**, which in turn is a sub class of class **Animal**:

```
1. class Animal {
2.     void eat() {
3.         System.out.println("This animal eats food.");
4.     }
5. }
6. 
7. class Dog extends Animal {
```

```
    void bark() {
        System.out.println("The dog barks.");
    }
}

class Puppy extends Dog {
    void weep() {
        System.out.println("The puppy weeps.");
    }
}
```

Objects of class **Animal** will have one method **eat()**. Objects of class **Dog** will have access to two methods, **bark()** and **eat()**, one from its own class and one from its base class. class **Puppy**, which is derived from class **Dog**, will have three methods, **weep()**, **bark()** and **eat()**, one from its own implementation and others from its base class.

Hierarchical inheritance

In hierarchical inheritance, multiple subclasses inherit from a single superclass. In this scenario, one class serves as a base class for multiple sub classes, forming a hierarchy. In the following example, **Animal** is a base class. Class **Dog** and class **Cat** are subclasses of **Animal** class. So, both the classes, **Dog** and **Cat** inheriting properties from the same base class.

```
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("The dog barks.");
    }
}

class Cat extends Animal {
    void meow() {
        System.out.println("The cat meows.");
    }
```

```
}

}
```

Objects of class **Dog** will have two methods **bark()** and **eat()**. objects of class **Cat** will also have two methods **meow()** and **eat()**.

Hybrid inheritance

Hybrid inheritance is a combination of two or more types of inheritance. Since Java does not support multiple inheritance through classes to avoid ambiguity, hybrid inheritance is typically achieved using interfaces.

An interface is a reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types. Interfaces cannot contain instance fields or constructors, and the methods in an interface are abstract by default unless they are defined with the default or static keyword. More about interfaces we are going to learn in *Chapter 5*, *Interface Abstraction and Packaging*.

Interfaces are powerful tools for achieving abstraction and multiple inheritance in Java. While a class can inherit from only one superclass (single inheritance), it can implement multiple interfaces, thus inheriting behavior from multiple sources. This allows for a more flexible design and the ability to create classes that can interact with multiple different systems or adhere to multiple different contracts. The following program demonstrates hybrid inheritance by using an interface **CanFly**:

```
interface CanFly {
    void fly();
}

class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

class Bird extends Animal implements CanFly {
    public void fly() {
```

```
        System.out.println("The bird flies.");
    }
}
```

Multiple inheritance

While inheriting, you can specify only one base class for any subclass. Java does not support multiple super classes for a single sub class, but it does allow a class to implement multiple interfaces, thereby achieving multiple inheritance of type. The following example demonstrates the concept of multiple inheritance:

```
interface CanFly {
    void fly();
}

interface CanSwim {
    void swim();
}

class Duck implements CanFly, CanSwim {
    public void fly() {
        System.out.println("The duck flies.");
    }

    public void swim() {
        System.out.println("The duck swims.");
    }
}
```

In this example program, two interfaces, **CanFly** and **CanSwim** are declared with methods **fly()** and **swim()** respectively. Class **Duck** implements both interfaces, so inherits behaviors from both interfaces.

'super' keyword

In the case of inheritance, the base class is also referred to as super class. The **super** keyword in Java is a reference variable used to refer to the immediate parent class object. It is commonly used in inheritance to access members (methods, fields, and constructors) of the superclass that are hidden by the subclass. In the next sections,

we will go through the detailed explanation of its uses and functionalities:

Accessing superclass methods

The super keyword can be used to call methods defined in the superclass. This is particularly useful when a method in the derived class is of the same name and type signature as in the base class. This is called method overriding. The subclass, in this scenario, has two copies of the method, one from its own class and another from base class. If we want to call the overridden method from the superclass, **super** keyword can be used.

```
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}
class Dog extends Animal {
    void eat() {
        System.out.println("The dog eats meat.");
    }
    void display() {
        super.eat(); // Calls the eat() method of Animal class
    }
}
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat(); // Output: The dog eats meat.
        dog.display(); // Output: This animal eats food.
    }
}
```

In the above program, the base class **Animal** has one method **eat()**. The subclass **Dog** overrides the **eat()** method. Now, it has two copies of **eat()** method, one local implementation and one from base class. In **display()** method, the base implementation of **eat()** method is invoked by using **super** keyword.

Accessing superclass fields

Sometimes the instance members of super class and sub class have the same name. The fields of **super** class will be hidden by fields of subclass. In this case, the super keyword can be used to access fields of the superclass. See the following example:

```
class Animal {
    String color = "White";
}

class Dog extends Animal {
    String color = "Black";

    void printColor() {
        System.out.println("Dog color: " + color); // Output: Dog color: B
        System.out.println("Animal color: " + super.color); // Output: Ani
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.printColor();
    }
}
```

In this program, the base class **Animal** and derived class **Dog** both have field **color**. Inside **printColor()** method, the first **System.out.println()** method call is printing local instance variable of **Dog** class while second statement is referencing the base class **color** by using super keyword.

Calling superclass constructors

The super keyword is used to invoke the constructor of the superclass. This is particularly useful when the superclass has a parameterized constructor, and the subclass needs to pass specific values to initialize the superclass. The syntax for subclass calling the constructor of base class is:

super(parameter_list);

The parameter list should be as per the specification given in the base class constructor. The **super()** statement must be the first

statement in the derived class constructor. Consider the following example:

```
class Animal {
    Animal(String name) {
        System.out.println("Animal constructor called. Name: " + name);
    }
}

class Dog extends Animal {
    Dog(String name) {
        super(name); // Calls the constructor of Animal class
        System.out.println("Dog constructor called. Name: " + name);
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog("Buddy");
        // Output:
        // Animal constructor called. Name: Buddy
        // Dog constructor called. Name: Buddy
    }
}
```

In this example program, **Dog** class is instantiated by passing the parameter **“Buddy”**, which invokes the **Dog** class constructor, with the name parameter initialized with **“Buddy”**. The Dog class constructor, in turn, invokes the base class (Animal) constructor with the name parameter. So, the program prints two lines:

Animal constructor called. Name: Buddy
Dog constructor called. Name: Buddy

One from the base class (**Animal**) constructor and another from the derived class (**Dog**) constructor.

Method overriding

Method overriding is a feature in object-oriented programming that allows a subclass to provide a specific implementation for a method that is already defined in its superclass. The overridden method in

the subclass must have the same name, return type, and parameters as the method in the superclass. When the subclass calls an overridden method, always local version of the method defined within the subclass will be referred to. The base class version of the method will be hidden. To get access to the base class implementation of the overridden method, **super** keyword can be used. (Refer to previous section)

Key concepts and rules of method overriding

The following are the key concepts and rules of method overriding:

- **Same method signature**: The method in the subclass must have the same name, return type, and parameters as the method in the superclass.
- **Access level**: The access level of the overriding method cannot be more restrictive than the overridden method. For example, if the superclass method is public, the subclass method cannot be protected or private.
- **Return type**: The return type of the overriding method must be the same or a subclass (covariant return type) of the return type declared in the superclass method.
- **Instance methods**: Only instance methods can be overridden. Static methods are not overridden but hidden if redefined in the subclass.
- **Final methods**: Methods marked as final cannot be overridden.
- **Constructors**: Constructors cannot be overridden.

Consider a simple example with an **Animal** superclass and a **Dog** subclass:

```
1. class Animal {
2.     void eat() {
3.         System.out.println("This animal eats food.");
4.     }
5. }
6. 
7. class Dog extends Animal {
8.     @Override
```

```
    void eat() {
        System.out.println("The dog eats meat.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();
        myAnimal.eat(); // Output: This animal eats food.

        Dog myDog = new Dog();
        myDog.eat(); // Output: The dog eats meat.

        Animal myPet = new Dog();
        myPet.eat(); // Output: The dog eats meat. (runtime polymorphism
    }
}
```

The **Dog** class overrides the eat method of the **Animal** class. The **@Override** annotation is used to indicate that a method is being overridden, which helps in preventing errors (e.g., if the method signature is incorrect).

In the **Main** class, an **Animal** reference is used to refer to a **Dog** object (**Animal myPet = new Dog();**). When the eat method is called on **myPet**, the **Dog** class's **eat** method is executed. This demonstrates runtime polymorphism, where the method to be executed is determined at runtime based on the object's actual type.

Polymorphism is one of the key concepts of OOP. The word polymorphism means having many forms. The word *poly* means many and *morphs* means forms.

In Java polymorphism is mainly divided into two types:

- Compile time polymorphism
- Runtime polymorphism

Compile time polymorphism is also known as **static polymorphism**. This type of polymorphism is achieved by overloading. The concept of overloading we covered in *Chapter 3, Building Classes*. Runtime polymorphism is also known as **dynamic method dispatch**. It is a process in which a function call to the

overridden method is resolved at Runtime. This type of polymorphism is achieved by method overriding.

Benefits of method overriding

The following are the benefits of method overriding:

- **Runtime polymorphism**: Overriding allows for runtime polymorphism, enabling the program to decide which method to invoke at runtime, enhancing flexibility.
- **Extensibility**: Subclasses can provide specific implementations for methods that are more appropriate for their context, allowing for better customization and extensibility.
- **Code reusability**: Superclass methods can be reused and extended in subclasses without modifying the existing code, promoting code reusability.

Dynamic method dispatch

Dynamic method dispatch, also known as **runtime polymorphism or late binding**, is a mechanism in Java that allows a method to be overridden in a subclass to be called through a reference of the superclass. This decision of which method to call is made at runtime based on the actual object that the reference points to. Dynamic method dispatch is a key aspect of polymorphism, allowing objects to be treated as instances of their parent class but still call the overridden methods in the subclass. It works through method overriding, where a subclass provides a specific implementation of a method that is already defined in its superclass.

Let us illustrate this with a concrete example involving a superclass **Animal** and two subclasses **Dog** and **Cat**:

```
1. // Superclass
2. class Animal {
3.     void makeSound() {
4.         System.out.println("Animal makes a sound");
5.     }
6. }
7.
8. // Subclass Dog
```

```
class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Dog barks");
    }
}

// Subclass Cat
class Cat extends Animal {
    @Override
    void makeSound() {
        System.out.println("Cat meows");
    }
}

// Main class to demonstrate dynamic method dispatch
public class Main {
    public static void dmdDemo(String[] args) {
        // Reference of superclass type pointing to subclass object
        Animal myAnimal;

        // Creating a Dog object and assigning it to myAnimal
        myAnimal = new Dog();
        myAnimal.makeSound(); // Calls the Dog's version of makeSound(

        // Creating a Cat object and assigning it to myAnimal
        myAnimal = new Cat();
        myAnimal.makeSound(); // Calls the Cat's version of makeSound()

        // Creating an Animal object and assigning it to myAnimal
        myAnimal = new Animal();
        myAnimal.makeSound(); // Calls the Animal's version of makeSou
    }
}
```

In this program, superclass **Animal** contains a method **makeSound()** that prints "**Animal makes a sound**". Subclass **Dog** overrides the **makeSound()** method to print "**Dog barks**". Subclass **Cat** overrides the **makeSound()** method to print "**Cat meows**". In the **Main** class **dndDemo** an **Animal** reference **myAnimal** is used to point to

different subclass objects (**Dog** and **Cat**). The actual method to be called is determined at runtime based on the object type that **myAnimal** refers to.

When you run the **Main** class, the output will be:

Dog barks
Cat meows
Animal makes a sound

The type of the reference variable (**myAnimal**) is **Animal**. The actual object that **myAnimal** refers to can be an instance of **Dog**, **Cat**, or **Animal**. When **myAnimal.makeSound()** is called, Java checks the actual object type at runtime and calls the overridden method in the corresponding subclass if it exists.

Covariant return type

Covariant return types were introduced in Java 5 to enhance the flexibility of method overriding. In simple terms, a covariant return type allows an overriding method in a subclass to return a type that is a subclass of the return type declared in the method of the superclass. This provides greater precision when returning objects from overridden methods and is particularly useful in object-oriented programming, where methods often need to return instances of their own class. This also ensures type safety and avoids the need for casting when using the returned objects. Covariant return types leverage polymorphism by allowing methods in subclasses to return more specific types. This enhances the expressiveness of the language and allows for more refined designs.

Consider a scenario with an **Animal** superclass and a **Dog** subclass:

```
1. class Animal {
2.     Animal getInstance() {
3.         return this;
4.     }
5. }
6.
7. class Dog extends Animal {
8.     @Override
9.     Dog getInstance() {
10.        return this;
```

```
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        Dog instance = dog.getInstance(); // Covariant return type
        System.out.println(instance instanceof Dog); // Output: true
    }
}
```

In this example, the **Animal** class defines a method **getInstance()** that returns an instance of **Animal**. The **Dog** class overrides the **getInstance()** method and specifies the return type as **Dog**, which is a subclass of **Animal**. This is allowed because of covariant return types.

When **dog.getInstance()** is called, it returns an instance of **Dog** without the need for any casting. This makes the code more type-safe and easier to read and maintain.

Consider a more complex hierarchy:

```
class Animal {
    Animal getInstance() {
        return this;
    }
}

class Mammal extends Animal {
    @Override
    Mammal getInstance() {
        return this;
    }
}

class Dog extends Mammal {
    @Override
    Dog getInstance() {
        return this;
    }
}

```

```
21. public class Main {
22.     public static void main(String[] args) {
23.         Mammal mammal = new Mammal();
24.         Mammal mammalInstance = mammal.getInstance(); // Returns a
25.
26.         Dog dog = new Dog();
27.         Dog dogInstance = dog.getInstance(); // Returns a Dog
28.
29.         // Polymorphism
30.         Animal animal = new Dog();
31.         Dog anotherDogInstance = ((Dog) animal.getInstance()); // Need
32.     }
33. }
```

In this example, each class in the hierarchy provides a more specific return type for the **getInstance()** method. This allows instances of the subclasses to be returned correctly without requiring casting, thus improving code safety and readability.

Role of final keyword in inheritance

The **final** keyword in Java can be used in various contexts, including classes, methods, and variables. When it comes to inheritance, the **final** keyword has specific implications. A **final** class cannot be extended by any other class. This is useful when you want to prevent inheritance to maintain the integrity of your class. The following example demonstrates the use of **final** keyword:

```
1. // Final class
2. final class Vehicle {
3.     String brand;
4.
5.     Vehicle(String brand) {
6.         this.brand = brand;
7.     }
8.
9.     void displayInfo() {
10.         System.out.println("Brand: " + brand);
11.     }
12. }
13.
```

```
// This will cause a compilation error because Vehicle is a final class
// class Car extends Vehicle {
//     Car(String brand) {
//         super(brand);
//     }
// }
```

In this example, attempting to subclass **Vehicle** will cause a compilation error because **Vehicle** is declared as final.

A **final** method cannot be overridden by subclasses. This is useful when you want to ensure that the implementation of a method remains unchanged in any subclass.

```
class Vehicle {
    String brand;

    Vehicle(String brand) {
        this.brand = brand;
    }

    // Final method
    final void displayInfo() {
        System.out.println("Brand: " + brand);
    }
}

class Car extends Vehicle {
    int mileage;

    Car(String brand, int mileage) {
        super(brand);
        this.mileage = mileage;
    }

    // This will cause a compilation error because displayInfo() is final in
    // @Override
    // void displayInfo() {
    //     System.out.println("Brand: " + brand + ", Mileage: " + mileage
    // }

    void displayMileage() {
```

```
        System.out.println("Mileage: " + mileage);
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car("Honda", 50000);
        myCar.displayInfo(); // Calls the final method from Vehicle
        myCar.displayMileage();
    }
}
```

In this example, attempting to override the **displayInfo()** method in the **Car** class will cause a compilation error because **displayInfo()** is declared as final in the **Vehicle** class.

A **final** variable cannot be reassigned once it has been initialized. This ensures the value remains constant throughout the life of the object.

```
class Vehicle {
    final String brand; // Final variable

    Vehicle(String brand) {
        this.brand = brand; // Final variable can be assigned once
    }

    void displayInfo() {
        System.out.println("Brand: " + brand);
    }
}

class Car extends Vehicle {
    int mileage;

    Car(String brand, int mileage) {
        super(brand);
        this.mileage = mileage;
    }

    void displayMileage() {
        System.out.println("Mileage: " + mileage);
```

```
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car("Honda", 50000);
        myCar.displayInfo();
        myCar.displayMileage();

        // This will cause a compilation error because brand is a final vari
        // myCar.brand = "Toyota";
    }
}
```

In this example, attempting to reassign the **brand** variable will cause a compilation error because the brand is declared as final in the **Vehicle** class.

Abstract class

Abstraction is a process of hiding the implementation details; only the functionality will be provided to the user. In other words, the user will have information on what the object does instead of how it does it. In Java, abstraction is achieved using abstract classes and interfaces.

An abstract class in Java is a class that cannot be instantiated on its own and is intended to be subclassed. It is used to provide a common definition of a base class that multiple derived classes can share. Abstract classes can have abstract methods, which are declared without an implementation, and concrete methods, which have an implementation. However, if a class has at least one abstract method, then the class must be declared abstract. Abstract class is declared by using **abstract** keyword. The general syntax of declaring an abstract class is:

abstract class A{
//abstract methods
// concrete methods
}

Key characteristics of abstract classes

An abstract class cannot be instantiated directly. It serves as a blueprint for other classes.

An **abstract** method is a method that is declared without an implementation. Subclasses are required to provide implementations for these methods. The general form of abstract method is: **abstract type methodName(parameter_list);**

- An abstract class can have methods with implementations (concrete methods) that can be inherited by subclasses.
- Abstract classes can have constructors, which can be called when a subclass is instantiated.
- Abstract classes can have fields (instance variables) that can be inherited by subclasses.
- Abstract classes can have static methods, which belong to the class rather than any instance of the class.
- Abstract classes can have final methods, which cannot be overridden by subclasses.

Let us see one example:

```
1. abstract class Animal {
2.     String name;
3.
4.     Animal(String name) {
5.         this.name = name;
6.     }
7.
8.     void sleep() {
9.         System.out.println(name + " is sleeping.");
10.    }
11.
12.    abstract void makeSound();
13. }
14.
15. class Dog extends Animal {
16.    Dog(String name) {
17.        super(name);
18.    }
19.
20.    @Override
```

```
    void makeSound() {
        System.out.println(name + " says: Woof Woof");
    }
}

class Cat extends Animal {
    Cat(String name) {
        super(name);
    }

    @Override
    void makeSound() {
        System.out.println(name + " says: Meow Meow");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog("Buddy");
        dog.sleep(); // Output: Buddy is sleeping.
        dog.makeSound(); // Output: Buddy says: Woof Woof

        Cat cat = new Cat("Kitty");
        cat.sleep(); // Output: Kitty is sleeping.
        cat.makeSound(); // Output: Kitty says: Meow Meow
    }
}
```

This program has four classes, abstract class **Animal**, subclasses **Dog** and **Cat**, and the **Main** class:

- **Abstract class Animal**: The class **Animal** is declared abstract using the **abstract** keyword. It has an abstract method **makeSound()**, which means it does not have an implementation in the **Animal** class. It also has a concrete method **sleep()** with an implementation. The **Animal** class has a constructor that initializes the name field.
- **Concrete subclasses Dog and Cat**: **Dog** and **Cat** are subclasses of **Animal** and must provide implementations for the **makeSound()** method. They use the **super(name)** call

in their constructors to initialize the name field inherited from the **Animal** class.

- **Main class**: In the **Main** class, instances of **Dog** and **Cat** are created. The **sleep()** method is called in both instances, which is inherited from the **Animal** class. The **makeSound()** method is called on both instances, which invokes the overridden implementations in the **Dog** and **Cat** classes.

When to use abstract classes

The following are the use case of abstract classes:

- **Common base class**: When you have a group of classes that share common characteristics and behaviors but also have their specific implementations.
- **Partial implementation**: When you want to provide a common implementation for some methods but leave other methods to be implemented by subclasses.
- **Default behavior**: When you want to provide a default behavior that can be overridden by subclasses if needed.
- **Shared state**: When you have state (fields) that is shared among multiple subclasses.

Conclusion

In this chapter, we have learned several crucial concepts of object-oriented programming in Java, focusing on inheritance and its related features. We explored different types of inheritance, such as single, multilevel, hierarchical, and hybrid, highlighting how each type facilitates code reuse and establishes a natural hierarchy among classes. We examined the super keyword, which plays a vital role in accessing superclass methods and fields, and calling superclass constructors, ensuring proper initialization and method access across class hierarchies. Method overriding was discussed in detail, emphasizing how it allows subclasses to provide specific implementations for superclass methods, enabling polymorphism and more dynamic code behavior.

The concept of covariant return types was introduced, showing how it allows for more specific return types in overridden methods, thus

enhancing the flexibility and type safety of the code. Finally, we explored abstract classes, which provide a common base for subclasses, allowing both abstract and concrete methods, and offering a blend of partial implementation and enforceable contracts for subclasses.

While discussing inheritance we introduced the interfaces. We are going to learn more about interfaces and also about packages in the next chapter.

Exercises

1. Create a base class Animal with fields for name and age. Write a method makeSound() that prints a generic sound. Create a derived class Dog that inherits from Animal. Override the makeSound() method to print "Bark". Create another derived class Cat that inherits from Animal. Override the makeSound() method to print "Meow". Create objects of Dog and Cat and call their makeSound() methods.
2. Create a base class Vehicle with fields make and model, and a method startEngine(). Create a derived class Car that adds a field numDoors and overrides startEngine() to print "Car engine started". Create another derived class ElectricCar that adds a field batteryLife and overrides startEngine() to print "Electric car engine started". Instantiate an object of ElectricCar and demonstrate method overriding.
3. Define a class Shape with a method draw() that prints "Drawing a shape". Define a subclass Circle that overrides draw() to print "Drawing a circle". Define another subclass Square that overrides draw() to print "Drawing a square". Create instances of Shape, Circle, and Square and call the draw() method on each object.
4. Create a base class Person with a method work() that prints "Person is working". Create a subclass Employee that overrides work() to print "Employee is working" but also calls the superclass work() method. Instantiate an Employee object and call its work() method to see the combined output.

5. Define an abstract class Shape with an abstract method calculateArea() and a non-abstract method display() that prints "Displaying shape". Create a subclass Rectangle that extends Shape, with fields width and height, and implements the calculateArea() method. Create another subclass Circle that extends Shape, with a field radius, and implements the calculateArea() method. Create objects of Rectangle and Circle, call their calculateArea() methods, and use the display() method.
6. Define an abstract class Vehicle with fields make and model, and an abstract method startEngine(). Provide a constructor to initialize make and model. Create a subclass Car that extends Vehicle and implements the startEngine() method. Create another subclass Bike that extends Vehicle and implements the startEngine() method. Instantiate objects of Car and Bike, and demonstrate the use of the constructor and the overridden method.
7. Create an abstract class Employee with fields name, id, and an abstract method calculateSalary(). Create a subclass SalariedEmployee with an additional field monthlySalary, and implement the calculateSalary() method. Create another subclass HourlyEmployee with additional fields hourlyRate and hoursWorked, and implement the calculateSalary() method. Instantiate objects of SalariedEmployee and HourlyEmployee, set their attributes, and call calculateSalary() for each to demonstrate polymorphism.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 5
Interface Abstraction and Packaging

Introduction

Abstraction is a core principle of **object-oriented programming (OOP)** that focuses on exposing only the essential features of an object while hiding the implementation details. This simplifies the complexity of software design by allowing developers to focus on high-level interactions without being bogged down by intricate details. Abstraction is achieved in Java through abstract classes and interfaces, which enable the creation of flexible and reusable code components.

On the other hand, packaging in Java refers to the organization of classes and interfaces into namespaces, known as packages. Packages are used to group related classes and interfaces, facilitating better organization, encapsulation, and namespace management. They help avoid naming conflicts, control access to classes and interfaces, and make it easier to locate and use related code components.

This chapter will discuss the intricacies of abstraction and

packaging in Java. Creating and using abstract classes we learnt in *Chapter 4, Inheriting Behaviour*. In this chapter, we will focus on interfaces. We will learn how to define and use interfaces to achieve abstraction, understanding their roles in designing flexible and extensible code structures. Additionally, we will cover the creation and utilization of packages, learning how to organize code logically, manage access levels, and maintain a clean and navigable codebase.

Structure

The chapter covers the following topics:

- Interfaces
- Packages

Objectives

This chapter aims to provide a comprehensive understanding of abstraction and packaging in Java. These are the fundamental concepts that help to enhance the design and organization of robust software systems. You will learn about interfaces and how to use it for achieving abstraction, simplifying complex systems by exposing only essential features while hiding implementation details. Additionally, the chapter will cover how to create and use packages to logically organize related classes and interfaces, manage access levels, and avoid naming conflicts.

Interfaces

An interface in Java is a reference type similar to a class, used to define what a class can do without specifying how it should be done. It can contain method signatures, constants, default methods, static methods, and nested types, but cannot include instance fields or constructors. By default, all methods in an interface are abstract unless they are static or default methods. Classes that implement an interface must provide implementations for all abstract methods. Java allows multiple inheritance by enabling a class to implement any number of interfaces. While an interface can declare constants, which are implicitly public, static, and final, implementing classes cannot modify them.

Since Java 8, interfaces can include default methods (with bodies), which can be overridden by implementing classes. They can also have static methods, which belong to the interface itself and can be accessed using the interface name. However, interfaces cannot be instantiated directly, as they do not have constructors.

Defining an interface

An interface can be defined in Java by using the **interface** keyword followed by the interface name. Abstract methods in an interface are declared without a body, meaning they do not have an implementation. Instead, they serve as a contract that any class implementing the interface must fulfill by providing concrete implementations for these methods. The general syntax of interface declaration is as follows:

```
interface interfaceName {
    returnType method1(parameterList);
    returnType method2(parameterList);
    ............
    type finalVariableName1 = value;
    type finalVariableName2 = value;
    ..........
}
```

Interfaces methods are implicitly public and abstract, so there is no need to specify these modifiers explicitly. Observe that the methods declared here are no body and ends with a semicolon. The variables are implicitly static and final, must be initialized with a constant value.

Let us see one basic example:

```
interface Animal {
    void eat();
    void sleep();
}
```

The **Animal** interface is defined with two abstract methods: **eat()** and **sleep()**.

Implementing an interface

To use an interface, a class is required to provide concrete

implementations for all the abstract methods declared in the interface. To implement an interface, a class uses the **implements** keyword followed by the interface name. The implementing class must define the body of each abstract method specified by the interface, thereby fulfilling the contract set by the interface. This allows the class to inherit the interface's method signatures and ensures that the class conforms to a particular behavior. Additionally, an implementing class can override default methods provided by the interface if custom behavior is needed. The following program demonstrates the implementation of **Animal** interface:

```
class Dog implements Animal {
    @Override
    public void eat() {
        System.out.println("The dog eats meat.");
    }

    @Override
    public void sleep() {
        System.out.println("The dog sleeps.");
    }
}
```

The **Dog** class implements the **Animal** interface. The **Dog** class must provide concrete implementations of all the abstract methods declared in the **Animal** interface. The **@Override** annotation is used to indicate that the methods are being overridden from the interface.

Now, this class can be instantiated and used. Let us write the **Main** class:

```
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();  // Output: The dog eats meat.
        dog.sleep(); // Output: The dog sleeps.

        Animal animal = new Dog();
        animal.eat();  // Output: The dog eats meat.
        animal.sleep(); // Output: The dog sleeps.
```

```
    }
}
```

In the **Main** class, an instance of **Dog** is created, and the **eat()** and **sleep()** methods are called on this instance. The **Animal** interface type reference (**Animal animal = new Dog();**) is used to demonstrate polymorphism, allowing the **Dog** instance to be treated as an **Animal**.

Interfaces can also include default methods, which come with a default implementation, allowing the implementing class to either use or override them. The following code demonstrates the default implementation of method:

```
interface Animal {
    void eat();
    void sleep();

    default void makeSound() {
        System.out.println("This animal makes a sound.");
    }
}

class Cat implements Animal {
    @Override
    public void eat() {
        System.out.println("The cat eats fish.");
    }

    @Override
    public void sleep() {
        System.out.println("The cat sleeps.");
    }

    // Override the default method
    @Override
    public void makeSound() {
        System.out.println("The cat says: Meow Meow");
    }
}
```

In this example, the **Animal** interface defines two abstract methods

(**eat()** and **sleep()**), and a default method (**makeSound()**). The implementing class must override abstract methods, while the default method may or may not be overridden by the implementing class.

If a class inherits from an interface but does not implement all its methods declared by that interface, in that case, the class must be declared as **abstract**. For example:

```
abstract class Cat implements Animal {
    @Override
    public void eat() {
        System.out.println("The cat eats fish.");
    }
}
```

An interface can have static methods also. Static methods in an interface are associated with the interface itself rather than any instance of the interface and can be called directly on the interface. Let us consider the following example:

```
interface Animal {
    void eat();
    void sleep();

    static void describe() {
        System.out.println("Animals are living beings that eat and sleep.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal.describe();  // Output: Animals are living beings that eat a
    }
}
```

The static method **describe()** of interface animal is directly called in **main()** method by using interface name. No object is required to call static method.

Constants defined in an interface are implicitly public, static, and final.

Benefits of using interfaces

Using interfaces in programming offers numerous benefits, enhancing flexibility, modularity, and maintainability of code:

- **Abstraction**: Interfaces allow you to define methods that must be implemented by any class that implements the interface, providing a clear contract.
- **Multiple inheritance**: A class can implement multiple interfaces, allowing a form of multiple inheritance and making it possible for a class to have more than one behavior.
- **Loose coupling**: Interfaces help to achieve loose coupling between components. Classes can interact with interfaces rather than specific implementations, making the system more flexible and easier to maintain.
- **Code reusability**: Interfaces promote code reusability by allowing different classes to implement the same set of methods defined by an interface.

Abstract class vs. interface

An abstract class and an interface are both constructs in object-oriented programming that allow for the definition of methods that must be implemented by derived classes, but they have distinct differences in terms of their purpose, usage, and capabilities.

An abstract class is a class that cannot be instantiated on its own and is meant to be subclassed. It can include both abstract methods, which have no implementation and must be implemented by subclasses, and concrete methods, which have an implementation and can be used as-is by subclasses. Abstract classes can also contain fields, constructors, and access modifiers. They provide a way to create a common base class with shared code that multiple derived classes can inherit from, allowing for code reuse and the establishment of a consistent interface. Abstract classes are suitable when there is a clear hierarchical relationship, where derived classes are a type of the base abstract class.

On the other hand, an interface is a contract that defines a set of methods (and properties in some languages) that a class must implement without providing any implementation itself. Interfaces cannot contain fields, constructors, or concrete methods (with some

modern exceptions allowing default methods in interfaces). They are used to define capabilities that can be implemented by any class, regardless of its position in the class hierarchy. This allows for a more flexible and decoupled design, enabling a class to implement multiple interfaces and thereby inherit behavior from multiple sources. Interfaces are particularly useful for defining plug-in capabilities, event handling, and ensuring a consistent method signature across different classes.

Extending an interface

An interface can be extended by another interface by using **extends** keyword. Extending an interface allows one interface to inherit the properties and methods of another interface. This mechanism promotes the reuse and modularization of code by enabling the creation of more specific interfaces based on more general ones. When an interface extends another, it inherits all the abstract methods of the parent interface, and it can also declare additional methods that the implementing classes must provide. This hierarchical relationship between interfaces is similar to class inheritance but with a focus on behavior rather than state or implementation. A class inheriting from an interface which extends from other interface must provide implementation to all the methods from both interfaces.

One significant benefit of extending interfaces is the ability to build complex and specialized types from simpler, more general types. For example, consider an interface Animal with a method **eat()**. A more specific interface, **FlyingAnimal**, can extend **Animal** and add a method **fly()**. Any class implementing **FlyingAnimal** will need to provide implementations for both **eat()** and **fly()**, ensuring that the class conforms to both the general **Animal** behavior and the specialized **FlyingAnimal** behavior. Let us see an example:

```
1. // General interface
2. interface Animal {
3.     void eat();
4. }
5. 
6. // Extended interface
7. interface FlyingAnimal extends Animal {
```

```
    void fly();
}

// Implementing the extended interface in a class
class Bird implements FlyingAnimal {
    @Override
    public void eat() {
        System.out.println("Bird is eating.");
    }

    @Override
    public void fly() {
        System.out.println("Bird is flying.");
    }
}

// Another class implementing the general interface
class Dog implements Animal {
    @Override
    public void eat() {
        System.out.println("Dog is eating.");
    }
}

// Main class to demonstrate the usage
public class interfaceExtendsDemo {
    public static void main(String[] args) {
        Bird bird = new Bird();
        bird.eat(); // Output: Bird is eating.
        bird.fly(); // Output: Bird is flying.

        Dog dog = new Dog();
        dog.eat(); // Output: Dog is eating.

        // Using polymorphism
        Animal myAnimal = bird;
        myAnimal.eat(); // Output: Bird is eating.
        // myAnimal.fly(); // This line would cause a compilation error

        FlyingAnimal myFlyingAnimal = bird;
```

```
        myFlyingAnimal.eat(); // Output: Bird is eating.
        myFlyingAnimal.fly(); // Output: Bird is flying.
    }
}
```

General interface **Animal** defines a method **eat()** which must be implemented by any class that implements this interface. Extended interface **FlyingAnimal** extends **Animal** and adds an additional method **fly()**. Any class implementing **FlyingAnimal** must implement both **eat()** and **fly()** methods. Class **Bird** implements the **FlyingAnimal** interface, thus providing implementations for both **eat()** and **fly()** methods. Class **Dog** implements the **Animal** interface and provides an implementation for the **eat()** method. It does not need to implement **fly()** because it does not extend **FlyingAnimal**. The main method of class **interfaceExtendsDemo** demonstrates creating instances of **Bird** and **Dog**, and shows how they use their respective methods. It also shows polymorphism, where a **Bird** object can be referenced as both an **Animal** and a **FlyingAnimal**.

Packages

Classes we have used so far in previous chapters are placed in the default namespace. Every time you create a new class, a unique name for the class should be selected. Otherwise, name conflict arises. This will lead to a situation when you will not be able to select conveniently a descriptive name for your class. you also need to check that the class name selected by you must not collide with class names given by other programmers. Package provides a solution for this problem. A package in programming, particularly in languages like Java, is a namespace that organizes a set of related classes and interfaces. Packages are used to group classes that are related in functionality, helping to modularize code and manage namespaces. This modularization makes large-scale software development more manageable, promotes code reuse, and helps prevent naming conflicts.

Packages provide a hierarchical namespace that organizes classes and interfaces. This prevents naming conflicts, especially in large projects where multiple classes might have the same name. For example, **java.util** and **java.sql** can both have a **Date** class, and

they are distinguished by their package names. Packages group related classes and interfaces together, making it easier to locate and manage code. For instance, utility classes can be grouped under **com.example.utils**, while database-related classes can go under **com.example.database**.

Defining a package

To create a package in Java, you use the **package** keyword at the top of your Java source file, followed by the package name. The package statement must be the first statement in the source code. This defines a namespace. Any classes declared within that file will be stored in the package. In the absence of the **package** statement, the declared class will go to the default package, which has no namespace. For small sample applications, the default package works fine. But while working with real-life applications, it is always advisable to work with packages. The general syntax for declaring a package is:

package pkgName;

Here, **pkgName** is the name given to the package. It should be a valid identifier. For example, to create a package named **myFirstPackage** the statement would be:

package myFirstPackage;

Packages are stored under file system folders. For the above package statement, there should be a folder named **myFirstPackage** in which the **.class** files of all the classes declared with the above package statement will be stored. The name of the folder should exactly match with the package name.

The same package statement can be used in more than one file. It simply mentions that the class belongs to that package.

You can also create a package hierarchy by separating the package names according to the hierarchy using dot(**.**). For example:

package pkg1.pkg2.pkg3;

Always remember to replicate the same directory structure in your file system. For example, a package declared with the statement:

package com.example.utils;

must be stored with a directory structure **com\example\utils** in

your Windows file system.

Integrated development environments (IDEs) like IntelliJ IDEA, Eclipse, and NetBeans provide tools to easily navigate and find packages in your project. They show a hierarchical view of your project structure. While using from command line, we need to take special care.

Let us see a simple example demonstrating the package concept in Java, focusing solely on basic functionalities. We will use package: **com.example.math**. The project is stored in **MyProject** directory. The directory structure is as follows:

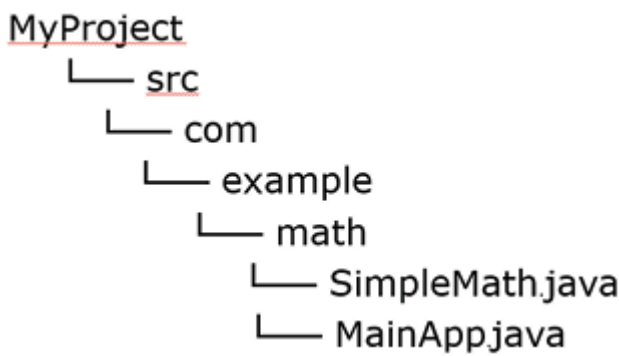


***Figure 5.1**: Directory structure, for example, package*

Let us create the classes in the **com.example.math** package:

```
1. // File: src/com/example/math/MainApp.java
2. package com.example.math;
3. class SimpleMath {
4.     public int add(int a, int b) {
5.         return a + b;
6.     }
7.     public int subtract(int a, int b) {
8.         return a - b;
9.     }
10. }
11. public class MainApp {
12.     public static void main(String[] args) {
13.         SimpleMath math = new SimpleMath();
14.         int sum = math.add(10, 5);
15.         int difference = math.subtract(10, 5);
```

```
16.
17.         System.out.println("Sum: " + sum);
18.         System.out.println("Difference: " + difference);
19.     }
20. }
```

To compile and run this example from the command line, follow these steps:

1. Navigate to the project directory:

   cd MyProject

1. Compile the classes:

   Use the **javac** command to compile the **MainApp.java** file. Specify the output directory for the compiled classes using the **-d** option.

   javac -d out src/com/example/math/MainApp.java

   javac is the Java compiler. **-d** out specifies the output directory (**out**). **src/com/example/math/MainApp.java** is the path to the source file. This command creates the **out/com/example/math** directory structure and places the compiled **SimpleMath.class** and **MainApp.class** files there.

1. Run the compiled program:

   Use the java command to run the **MainApp** class. Specify the classpath using the **-cp** option, pointing to the **out** directory where the compiled classes are located.

   java -cp out com.example.math.MainApp

   java is the Java runtime launcher. **-cp out** specifies the classpath, pointing to the out directory. **com.example.math.MainApp** is the fully qualified name of the class to run.

Importing a package

If the classes exist in the same package as our previous example, they can be used directly. But if the class which we want to use in our program exists in another package we need to import it to make it available. To use a class from a package, you need to import it using the **import** statement. In Java, importing packages allows you to access classes, interfaces, and other types defined in those

packages without needing to use their fully qualified names. Once a package is imported, we can access the class by using its name only.

In the Java source file, the import statement must be written on the top, immediately following the package statement (if any), before the declaration of any class. The general syntax is as follows:

import pkgName.className;

or import pkg1.pkg2.className;

This is called a single import. In this case, one class is imported:

or import pkg1.pkg2.*;

This is a wildcard import. You can import all the classes from a package using the wildcard *.

For example, **import java.util.*;** statement will import all classes and interfaces from **java.util** package. While **import java.util.Date;** statement will import **Date** class from **java.util** package.

We can write basic Java programs without importing any package. All the classes that are essential for Java programming are kept in the **java.lang** package. It is automatically imported for all programs by the Java compiler; you do not need to import this package explicitly. For using classes declared in other packages you need to write import statements. In Java, the primary way to bring classes from other packages into your code is by using the **import** statement. However, instead of importing a class, you can use its fully qualified name directly in your code. Let us consider the following program.

```
1. import java.util.Date;
2. class MyClass
3. {
4. public static void main(String arg[])
5. {
6. System.out.println(new Date());
7. }
8. }
```

In this program **java.util.Date** class is imported to print the current date. The same task can be achieved without **import** statement, as follows:

```
1. class MyClass
2. {
3. public static void main(String arg[])
4. {
5. System.out.println(new java.util.Date());
6. }
7. }
```

Package naming conventions

Package naming conventions play a crucial role in organizing and maintaining clean, understandable code, especially in large-scale software projects. Some commonly used naming conventions are listed below:

- **Domain-based naming**: It is common to use reverse domain names for package names to ensure uniqueness. For example, if your organization's domain is **example.com**, you might use **com.example** as the base package name.
- **Descriptive names**: Package names should be descriptive of the functionality they encapsulate. For example, **com.example.database** for database-related classes.
- **Standard library packages**: Java provides a rich set of standard library packages. Some of the commonly used packages include:
    - **java.lang**: Fundamental classes like String, System, and Math.
    - **java.util**: Utility classes like Collections, List, and Date.
    - **java.io**: Input and output classes, including File and InputStream.
    - **java.net**: Classes for networking, such as Socket and URL.

It is also essential to avoid names that conflict with standard library packages, like **java.util**, to prevent confusion and integration issues.

Following these conventions results in a more organized, maintainable codebase and reduces the risk of naming conflicts, particularly when integrating external libraries.

In *Chapter 3, Building Classes,* we discussed about Java access control mechanism. Package adds another dimension to it. Packages help define access controls. The four access levels in Java (public, protected, package-private, and private) determine how classes and members are accessible and up to which level.

The public member is accessible from any other class. The protected member is accessible within its own package and by subclasses. The member, without any modifier (package-private or default), is accessible only within its own package. The private member is accessible only within its own class.

Observe the following code where the class is declared with a public access modifier:

```
// File: src/com/example/foo/Foo.java
package com.example.foo;
public class Foo {
    public void display() {
        System.out.println("Public method in Foo");
    }
}
// File: src/com/example/bar/Main.java
package com.example.bar;

import com.example.foo.Foo;

public class Main {
    public static void main(String[] args) {
        Foo foo = new Foo();
        foo.display();  // Accessible because display() is public
    }
}
```

In this example, public class Foo is declared inside package **com.example.foo**. It is assessable inside **Main** class, which is in another package, **com.example.bar**.

Now, let us look at another example to understand **protected** members, which can be accessed within the same package and by subclasses.

```
// File: src/com/example/foo/Foo.java
package com.example.foo;
```

```
3.
4. public class Foo {
5.     protected void display() {
6.         System.out.println("Protected method in Foo");
7.     }
8. }
9. // File: src/com/example/foo/Bar.java
10. package com.example.foo;
11. public class Bar extends Foo {
12.     public void show() {
13.         display();  // Accessible because Bar is in the same package and is
14.     }
15. }
16. // File: src/com/example/bar/Main.java
17. package com.example.bar;
18.
19. import com.example.foo.Bar;
20.
21. public class Main {
22.     public static void main(String[] args) {
23.         Bar bar = new Bar();
24.         bar.show();  // Calls the protected method through a public metho
25.     }
26. }
```

In this example, the public class Foo in **com.example.foo** has one protected method **display()**. This class is subclassed by **Bar** class of same package. The public **show()** method of this class is calling **display()** method. Here, it is accessible because both class belongs to the same package, and **Bar** class is a subclass of the **Foo** class. The **Main** class is part of another package **com.example.bar** cannot directly access the protected method of the **Foo** class, but it can call the protected method through the public method shown in the **Bar** class.

In case of no modifier, also called a package-private modifier, the class will be assessable within the same package:

```
1. // File: src/com/example/foo/Foo.java
2. package com.example.foo;
3.
4. class Foo {
```

```
    void display() {
        System.out.println("Package-private method in Foo");
    }
}

// File: src/com/example/foo/Bar.java
package com.example.foo;

public class Bar {
    public void show() {
        Foo foo = new Foo();
        foo.display();  // Accessible because Bar is in the same package as
    }
}

// File: src/com/example/bar/Main.java
package com.example.bar;

import com.example.foo.Foo;  // This will cause a compile-
time error because Foo is package-private

public class Main {
    public static void main(String[] args) {
        // Foo foo = new Foo();  // Not accessible
        // foo.display();  // Not accessible
    }
}
```

In this example program, class **Foo** is declared without any modifier. So, it is available in **com.example.foo** package with package-private access. Only the members of this package can access this class. We cannot access this class outside the package.

Private members of a class are accessible inside the class only. Consider the following program:

```
// File: src/com/example/foo/Foo.java
package com.example.foo;

public class Foo {
    private void display() {
        System.out.println("Private method in Foo");
```

```
    }

    public void show() {
        display();  // Accessible within the same class
    }
}

// File: src/com/example/foo/Bar.java
package com.example.foo;

public class Bar {
    public void test() {
        Foo foo = new Foo();
        // foo.display();  // Not accessible because display() is private
        foo.show();  // Accessible because show() is public
    }
}
```

The public class **Foo** have two methods, a private **display()** method and a public **show()** method. The class **Bar** can access the **show()** method but will not be able to call private **display()** method.

Let us summarize the different scenarios:

Public						
Protected						
Package-private (default, no modifier)						
Private						

***Table 5.1**: Summary of different access levels*

Conclusion

In this chapter, we explored the concepts of interfaces and packages in Java. Interfaces allow you to define a contract that classes can implement, ensuring a consistent API and promoting polymorphism. We covered how to define an interface, implement it in a class, and the distinctions between interfaces and abstract classes. Additionally, we examined how interfaces can be extended to create more specialized contracts. Packages provide a mechanism for organizing classes and interfaces into namespaces, helping manage large codebases more effectively. We learned how to define a

package and the process of importing packages to utilize their classes and interfaces in different parts of your application. We have also discussed different access level scenarios with the help of various examples.

In the next chapter, we will discuss exceptions and how they can be handled in java programming.

Exercises

1. Define an interface Animal with methods eat() and sleep(). Create two classes, Dog and Cat, that implement this interface. Write a Main class to create instances of Dog and Cat, and call their eat() and sleep() methods.
2. Define an interface Vehicle with a default method start() that prints "Vehicle is starting". Create a class Car that implements Vehicle and override the start() method to print "Car is starting". Write a Main class to demonstrate calling the start() method on an instance of Car.
3. Define two interfaces, Flyable with a method fly() and Swimmable with a method swim(). Create a class Duck that implements both Flyable and Swimmable. Write a Main class to create an instance of Duck and call both fly() and swim() methods.
4. Define an interface Calculator with a static method add(int a, int b) that returns the sum of two numbers. Write a Main class to demonstrate calling the add() method of Calculator.
5. Create a package com.example.utils with a class StringUtils that has a static method reverse(String s) to reverse a string. Create a Main class in the com.example.main package that uses StringUtils to reverse a string and prints the result.
6. Create a package com.example.shapes with a class Circle that has a protected method calculateArea(). Create a subclass ColoredCircle in the same package that overrides calculateArea(). Create another class ShapeTester in a different package com.example.test that tries to access calculateArea() on an instance of ColoredCircle.
7. Create a package com.example.data with a class Person that

has a package-private field name and a method getName(). Create a Main class in the same package that creates an instance of Person and prints the name. Move the Main class to a different package and observe the compilation errors. Modify the code to access name correctly.

8. Create a package com.example.operations with a class MathOperations that has a method multiply(int a, int b) to multiply two integers. Write a Main class in another package that uses MathOperations without importing it (use the fully qualified name).

Chapter 6
Exception Handling

Introduction

An exception is an issue that occurs while a program is running. When an exception happens, it disrupts the usual flow of the program and can cause it to terminate unexpectedly, which is not recommended. Therefore, these exceptions are to be handled. An exception can occur for many different reasons. Some scenarios where an exception can occur are when a user has entered invalid data, a file that needs to be opened cannot be found, a network connection has been lost in the middle of communications, or the **Java Virtual Maching (JVM)** has run out of memory. Such abnormal situations need to be handled for robust, reliable, and maintainable software development. Exception handling is a method used to manage runtime errors like ClassNotFoundException, IOException, SQLException, RemoteException, and others. This chapter deals with how to handle exceptional situations in Java programs through exception-handling mechanisms. Topics such as try-catch blocks and best practices in exception handling are covered to ensure comprehensive error management.

Structure

The chapter covers the following topics:

- Exception class
- Exception handling blocks
- Built-in checked and unchecked exceptions
- User-defined exceptions
- Best practices for exception handling in Java
- Centralize error codes and descriptions

Objectives

By the end of this chapter, readers will understand the exception class hierarchy, including the role of the Exception class within the broader Throwable hierarchy, and be able to differentiate between exceptions and errors. They will identify and use built-in checked and unchecked exceptions, applying appropriate handling techniques in various scenarios. Additionally, readers will learn to create and utilize user-defined exceptions to enhance code readability and error management. The chapter will cover the effective use of exception handling constructs such as attempt block (try), exception capture (catch), raise exception (throw), exception declaration (throws), final block (finally), ensuring clear and efficient error management. Finally, readers will adopt best practices in exception handling, including strategies for logging and rethrowing exceptions, ensuring resource management and application stability, and creating informative custom exceptions for better error communication and debugging. Through these objectives, readers will be well-prepared to handle exceptions in their Java applications, leading to more reliable and user-friendly software.

Exception class

The Exception class is a fundamental component of Java's error handling mechanism, representing conditions that a sensible application might need to handle. It is part of the broader Throwable hierarchy, which is the superclass for all errors and

exceptions in Java.

In the Throwable hierarchy, there are two main subclasses: Error and exception. The following figure shows the hierarchy:

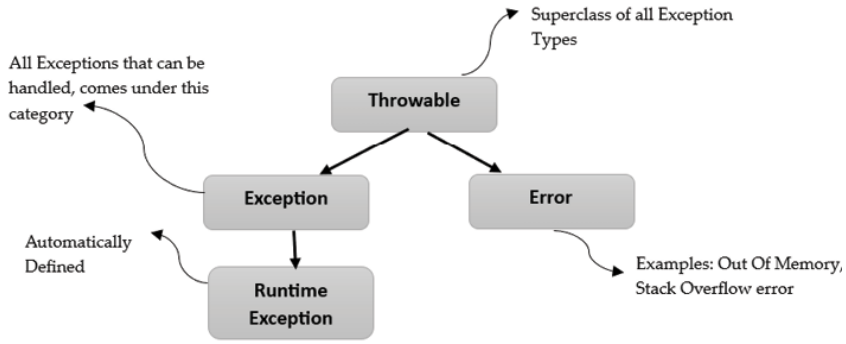


***Figure 6.1**: Exception class hierarchy*

Exception class is for exceptional conditions that the program should catch. All exception types are subclasses of class *Throwable*, which is at the top of exception class hierarchy. This class is extended to create user-specific exception classes. **RuntimeException** is a subclass of **Exception**. Exceptions under this class are automatically defined for programs. Exceptions of type **Error** are used by the Java run-time environment to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error. Error represents critical issues that a well-designed application should not attempt to handle. Errors are typically external to the application and indicate problems such as memory overflow or hardware failure.

An exception is an occurrence that interrupts the regular sequence of a program's instructions. It can be generated by the JVM or can be manually generated by code. When an exceptional condition occurs in the flow of the program during execution, an object representing the error condition is created and thrown in the method where it was caused. Exceptions arise generally due to violation of rules of Java language or constraints of the Java execution environment. When an exception arises in a method, the method can choose to handle it or can pass it on to the **caller** method. This process continues until the exception is caught or reaches the **main** method. If uncaught, it causes the program to

terminate.

Let us see one simple example program which contains one expression to print the value of an array element:

```
1. public class UncheckedExceptionDemo {
2.     public static void main(String[] args) {
3.         int[] data = {1, 2, 3};
4.         System.out.println(data[6]); // This will throw ArrayIndexOutOfB
5.     }
6. }
```

This program is intentionally trying to print the sixth element of the array, which is not available, to demonstrate the error condition. When the Java run time encounters the attempt to access the array element which does not exist, it creates an **Exception** object and hands it over to JVM. This process is known as throwing an exception. The exception object contains the details of the exception and the current status of the program where the exception occurred. The run-time system searches for an exception handler, which is not provided in the above program. The runtime system then transfers the Exception object to the default exception handler, a component of the runtime system. This handler outputs the exception details in a specific format and ends the program unexpectedly.

When the above listed program runs, the attempt to access **data[6]** will cause an **ArrayIndexOutOfBoundsException**. Since there is no catch block to handle this exception, the default exception handler is invoked, and the output will be something like:

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 6
at UncheckedExceptionDemo.main(UncheckedExceptionDemo.java:4)

It prints the stack trace of the exception to the standard error stream (usually the console). The stack trace includes the type of exception, an optional message associated with the exception, and the sequence of method calls (stack frames) that led to the exception. A stack trace is a report of the active stack frames at a certain point in time during the execution of a program. It provides a snapshot of the call stack (the hierarchy of method calls) at the moment an exception is thrown. Stack traces are useful for debugging because they show the sequence of method calls that led

to the error, helping developers identify where and why the error occurred.

This string is providing information about the specific type of exception, which is **ArrayIndexOutOfBoundsException** (a subclass of **Exception** class), in this case, with detail description. Observe the class name followed by method name (**main**), filename (**UncheckedExceptionDemo.java**) and the line number (**4**) where the exception has occurred.

Exception handling blocks

The default exception handler provided by the Java runtime system is quite useful for debugging, but sometimes, it is rather too technical and may be insufficient. You need to handle exception to ensure that the program can handle exceptional conditions gracefully. Java uses five keywords to handle exceptions. The **try**, **catch**, **finally**, **throw**, and **throws** keywords are essential components of exception handling in Java. The **try** block encompasses code that could potentially throw an exception, effectively safeguarding this section of code known as the guarded region. A **catch** statement specifies the type of exception it aims to handle. When an exception arises in the guarded code, the corresponding catch block is inspected. If the exception type matches, control is passed to the **catch** block, which then manages the exception. Regardless of whether an exception was thrown, the finally block executes, often containing cleanup code. The throw statement is used to explicitly trigger an exception. The throws clause appears in method signatures to indicate that the method may throw specified exceptions.

The general syntax of exception handling is as follows:

```
try {
   // Code that might throw an exception
}
catch (ExceptionType name) { // ExceptionType is the type of exception that has occurred.
   // Code to handle the exception
}
finally {
```

```
    // Code that will always run
}
```

When an exception arises in the **try** block, control promptly shifts to the nearest **catch** block that matches the exception type. If a corresponding **catch** block is identified, it handles the exception. If no suitable **catch** block is located, the exception is passed up to the **calling** method. Regardless of whether an exception is thrown, the **finally** block runs after the **try** and **catch** blocks.

Handling Exceptions with try..catch

In Java, the **try** and **catch** blocks work together to manage exceptions. The **try** block is used to wrap the code that might throw an exception. It defines a block of statements that will be monitored for exceptions. This includes method calls, operations that can cause runtime exceptions, and code that interacts with external resources like files, databases, or networks.

The **catch** block is designed to manage exceptions thrown by the code within the **try** block. Each **catch** block acts as an exception handler, dealing with the specific type of exception indicated by its parameter. Multiple **catch** blocks can be used to handle various types of exceptions individually, with each **catch** block addressing a particular type of exception. To illustrate the concept, let us see a program to process **ArrayIndexOutOfBounds** exception:

```
1. public class ExceptionHandling{
2.     public static void main(String[] args) {
3.         int[] data = {10, 20, 30}; // An array of length 3
4.
5.         try {
6.             // Attempting to access index 5 which is beyond available
   index
7.             System.out.println(data[5]);
8.         } catch (ArrayIndexOutOfBoundsException e) {
9.             // Handling the exception
10.    System.out.println("Attempt to access array
   index out of bounds: " + e.getMessage());
11.        }
12.
13.        // Continue with the rest of the program
```

```
14.         System.out.println("Program continues after handling the exceptio
15.     }
16. }
17.
```

If you run the above program, the output will be:

Attempt to access array index out of bounds: Index 5 out of bounds for length 3
Program continues after handling the exception.

The statement accessing an invalid array index is placed inside the **try** block. It causes the specific **ArrayIndexOutOfBoundsException**, which is thrown by the code. The **catch** block intercepts the exception and manages it by displaying a user-friendly message that includes the exception's details. The **e.getMessage()** method provides a description of the exception. After the exception is handled, the program continues to execute the subsequent statements. This demonstrates that proper exception handling allows the program to recover gracefully and continue its execution. A **try** and its **catch** block form a unit. The code that might throw an exception should be protected by **try** and must be enclosed within curly braces, essentially marking a scope where the program should be monitored for potential runtime issues. If an exception happens within the **try** block, the control immediately transfers to the associated **catch** block(s). Each catch block is equipped to handle a specific type of exception, providing the means to manage different error conditions separately. The **catch** blocks are scoped to handle only exceptions thrown within the associated **try** block, creating a cohesive unit where error-prone code and its handling logic are closely linked. This structure ensures that exceptions are caught and managed locally, allowing the program to maintain control flow and perform necessary cleanup or recovery actions.

Statements in a **try** block may go through more than one exception. In such situations, multiple **catch** blocks can be written to handle different types of exceptions. When an exception is thrown, the JVM searches for the first **catch** block that can handle the specific exception type, and control is transferred to that block. If one **catch** block is executed, others are bypassed, and execution continues after the *try-catch* block. If none of the **catch** blocks can handle the

exception, it propagates up the call stack. This mechanism allows developers to tailor their error handling to different scenarios, providing specific responses to distinct exceptions. For instance, if a **try** block includes code that could potentially throw either an **ArrayIndexOutOfBoundsException** or a **FileNotFoundException**, two **catch** blocks can be used to handle these exceptions separately. Let us see one example of handling two different exceptions within a **try** block:

```
1. public class MultipleCatchDemo {
2.     public static void main(String[] args) {
3.         try {
4.             // Attempt to access an array element that may be out of bound
5.             int[] data = {1, 2, 3};
6.             System.out.println("Accessing element at index 5: " + data[5]);
7.
8.             // Attempt to perform division that may result in an arithmetic
9.             int result = 10 / 0;
10.            System.out.println("Result of division: " + result);
11.        } catch (ArrayIndexOutOfBoundsException e) {
12.            System.out.println("Error: Attempted to access an invalid array
13.        } catch (ArithmeticException e) {
14.            System.out.println("Error: Arithmetic issue encountered. " + e.g
15.        }
16.
17.        // Continue with the rest of the program
18.        System.out.println("Program continues after handling the exceptio
19.    }
20. }
```

The code **data[5]** attempts to access the element at index **5** of an array with only three elements, which throws an **ArrayIndexOutOfBoundsException**. The code 10/0 attempts to divide 10 by zero, which throws an **ArithmeticException**. The first **catch** block is written to catch **ArrayIndexOutOfBoundsException**. Second **catch** block is provided to catch **ArithmeticException**. Whichever exception is thrown during program execution, the corresponding **catch** block would be matched by the runtime system to handle the exception. After handling the exceptions, the program continues to execute the remaining statements. The program will give the following output:

Error: Attempted to access an invalid array index. Index 5 out of bounds for length 3
Program continues after handling the exceptions.

The output shows that the **ArrayIndexOutOfBoundsException** was caught and handled, but the **ArithmeticException** was not reached in this particular run because the program terminated at the first exception encountered in the **try** block.

If the code were structured to ensure both exceptions could potentially be thrown (e.g., by separating the two risk-prone operations), both catch blocks would demonstrate their handling capabilities.

When writing multiple **catch** blocks, the order in which you catch exceptions matters. **catch** blocks should be arranged from the most specific to the most general. This is because the Java compiler checks each **catch** block in sequence, and once a match is found, it skips the remaining blocks. Catching a general exception type first would prevent any more specific exception types from being caught. Let us see one program:

```
1. public class MultipleCatchDemo {
2.     public static void main(String[] args) {
3.         try {
4.             int[] data = {10, 20, 30};
5.             System.out.println("Accessing element at index 5: " + data[5]);
6.
7.             int result = 10 / 0; // This can throw ArithmeticException
8.             System.out.println("Result of division: " + result);
9.         } catch (ArrayIndexOutOfBoundsException e) {
10.            System.out.println("Error: Array index out of bounds. " + e.getl
11.        } catch (ArithmeticException e) {
12.            System.out.println("Error: Arithmetic issue. " + e.getMessage()
13.        } catch (Exception e) {
14.            System.out.println("Error: General exception occurred. " + e.ge
15.        }
16.
17.        System.out.println("Program continues after handling the exceptio
18.    }
19. }
```

The first catch block catches **ArrayIndexOutOfBoundsException**, a

specific runtime exception. This block handles the situation where an array is accessed with an invalid index. The second **catch** block catches **ArithmeticException**, another specific runtime exception. This block handles arithmetic errors such as division by zero. The last catch block catches any Exception that was not caught by the previous blocks. This acts as a fallback to handle any other exceptions that may arise. By placing it last, you ensure it does not preempt more specific exception handling.

If you reverse the order of the **catch** blocks and place the general Exception **catch** block first, the compiler will produce an error. This is because the more specific **catch** blocks would become unreachable, as the general Exception block would catch all exceptions, making the specific blocks redundant. Let us see one example of incorrect order:

```
1. // This will cause a compile-time error.
2. public class IncorrectCatchOrderDemo {
3.     public static void main(String[] args) {
4.         try {
5.             int[] data = {10, 20, 30};
6.             System.out.println("Accessing element at index 5: " + data[5]);
   throws ArrayIndexOutOfBoundsException
7.
8.             int result = 10 / 0; // throws ArithmeticException
9.             System.out.println("Result of division: " + result);
10.        } catch (Exception e) {
11.            System.out.println("Error:
   General exception: " + e.getMessage());
12.        } catch (ArrayIndexOutOfBoundsException e) {
13.            System.out.println("Error: Array index out of
   bounds. " + e.getMessage());
14.        } catch (ArithmeticException e) {
15.            System.out.println("Error: Arithmetic issue.
   " + e.getMessage());
16.        }
17.
18.        System.out.println("Program continues after handling the exceptio
19.    }
20. }
```

If you try to compile this program, it will give an error message

stating that the Exception has already been caught.

A **try** block can be placed inside another **try** or **catch** block. This is useful in complex scenarios where different parts of the code require different handling mechanisms or when you need to manage multiple levels of error-prone operations. Nested **try** blocks enable handling exceptions specific to certain operations independently within a larger context. This allows finer control over error management. Here is an example of nesting of try blocks:

```
1. public class NestedTryCatch {
2.     public static void main(String[] args) {
3.         try {
4.             // Outermost try block
5.             System.out.println("Beginning of outer try block");
6.
7.             try {
8.                 // First Inner try block
9.                 System.out.println("Beginning of first
inner try block");
10.                int result = 10 / 0; // This triggers ArithmeticException
11.                System.out.println("End of first inner try block ");
12.            } catch (ArithmeticException e) {
13.                System.out.println("ArithmeticException caught: " + e.getMe
14.            }
15.
16.            try {
17.                // Second Inner try block
18.                System.out.println("Beginning of second inner try block");
19.                int[] array = new int[5];
20.                array[10] = 50; // This triggers ArrayIndexOutOfBoundsExc
21.                System.out.println("End of second inner try block");
22.            } catch (ArrayIndexOutOfBoundsException e) {
23.                System.out.println("ArrayIndexOutOfBoundsException caugh
24.            }
25.
26.            System.out.println("End of outer most try block");
27.        } catch (Exception e) {
28.            System.out.println("Exception caught in outer most try block: "
29.        }
30.
```

```
31.         System.out.println("Program continues...");
32.     }
33. }
```

In the above program, the outer **try** block contains two inner **try** blocks and a **catch** block to manage any exceptions not caught by the inner **try** blocks. The first inner try block attempts to divide by zero, which throws an **ArithmeticException**, and is managed by the appropriate **catch** block, which prints an error message. The second inner **try** block attempts to access an invalid array index, which throws an **ArrayIndexOutOfBoundsException**. This exception is managed by the appropriate **catch** block, which prints an error message. The outer **catch** block catches any exceptions not handled by the inner try blocks (in this example, it will not catch any because both inner exceptions are caught by their respective blocks). The program will give the following output:

Beginning of outer try block
Beginning of first inner try block
ArithmeticException caught: / by zero
Beginning of second inner try block
ArrayIndexOutOfBoundsException caught: 10
End of outer most try block
Program continues...

throw clause

In the previous section, we have learned to deal with exceptions raised by the Java run time environment. In some cases, we may need to raise an exception explicitly to report an exceptional condition. This is very crucial in situations where a programmer wishes to create a customized error condition and raise an alarm when a certain condition is met. To explicitly generate an exception in Java, **throw** keyword is used, which can be a newly instantiated exception or a pre-existing one. This mechanism allows developers to create and handle error conditions deliberately within a program. When an exception is thrown, the normal flow of the program is disrupted, and control is transferred to the nearest enclosing **catch** block matching the type of the exception. The general syntax of a throw statement is:

throw throwableObject;

The **throwableObject** must be an object of Throwable or of its subclasses.

The program executes normally until it encounters a throw statement. When a throw statement is executed, an exception object is created and thrown. This object must be of a type that inherits from the **Throwable** class (typically **Exception** or **RuntimeException**).

Once the exception is thrown, the method from which the exception is thrown stops executing at the point of the **throw** statement. The call stack is searched by a runtime system for a **catch** block that can handle the type of thrown exception. This search begins with the current method. If the current method does not have a suitable **catch** block, the runtime system terminates the method and returns to the method that called it. This process continues, unwinding the stack, until a **catch** block is found that can handle the thrown exception. When an appropriate **catch** block is found, control is transferred to that block. The **catch** block then executes, handling the exception. After the **catch** block has been executed, the program resumes execution after the try-catch block, continuing normally unless another exception is thrown. If no suitable **catch** block is found by the time the runtime system reaches the top of the call stack, the program terminates, and the default exception handler handles the uncaught exception, typically by printing a stack trace and an error message. Here is one example demonstrating the process:

```
1. public class ThrowDemo {
2.     public static void main(String[] args) {
3.         try {
4.             validateAge(15); // Step 1: Normal execution
5.         } catch (IllegalArgumentException e) { // Step 6: Catch block foun
6.             System.out.println("Error: Invalid
   Age:" + e.getMessage()); // Handling exception
7.         }
8.         // Step 7: Resume execution here if no exception or after catch blo
9.     }
10.
11.    static void validateAge(int age) {
12.        if (age < 18) {
```

```
13.             throw new IllegalArgumentException("Age should be 18 or mor
14.         } else {
15.             System.out.println("Welcome!
    Access granted.»); // Normal execution
16.         }
17.         // Step 3: Execution stops here if exception is thrown
18.     }
19. }
```

In this example, when **validateAge(15)** is called, the condition **age < 18** is true, so an **IllegalArgumentException** is thrown. Observe the creation of **IllegalArgumentException** object by using a **new** keyword. The exception immediately interrupts the normal flow, and control is transferred back to the main method. The runtime system looks for a **catch** block in **main()** method that can handle **IllegalArgumentException**. It finds the matching **catch** block that handles the exception, prints the message, and then normal execution resumes after the try-catch block.

throws

In Java, any method that might cause exceptions must specify all the possible exceptions it could throw. This way, anyone using the method is informed in advance about the exceptions that need to be managed. The **throws** keyword is used in method signatures to indicate that a method may throw one or more exceptions. This declaration informs the compiler and the calling code about the potential exceptions that need to be handled or propagated further up the call stack. The **throws** clause is particularly useful for checked exceptions, which must be either caught or declared in the method signature. By using **throws**, a method acknowledges that it might encounter specific error conditions that it does not handle internally, thereby delegating the responsibility of handling these exceptions to the calling method. This approach promotes cleaner and more robust error handling, making it clear which methods might fail and ensuring that exceptions are properly managed. The general syntax of method declaration with **throws** keyword is as follows:

type methodName(parameterList) throws exceptionList
{ //definition of method }

The **parameterList** and **exceptionList** are comma separated lists of parameters and exceptions, respectively.

If the **throws** keyword is not used in a program where a method can potentially throw a checked exception, the code will not compile. The Java compiler enforces that any method that might throw a checked exception must either handle it using a try-catch block or declare it with the **throws** keyword. On the other hand, unchecked exceptions (subclasses of **RuntimeException**), do not need to be declared or caught.

Here is an example illustrating what happens if the **throws** keyword is not used for a checked exception:

```
1. public class BasicThrowsExample {
2.     public static void main(String[] args) {
3.         try {
4.             validateAge(15); // This will throw a compile-
   time error
5.         } catch (Exception e) {
6.             System.out.println("Error: Invalid
   Age: " + e.getMessage());
7.         }
8.     }
9.
10.    // This method might throw an Exception but does not declare it wit
11.    static void validateAge(int age) {
12.        if (age < 18) {
13.            // Throwing a checked exception
14.            throw new Exception("Age should be 18 or more"); // Compile-
   time error
15.        }
16.        System.out.println(" Got the Access!");
17.    }
18. }
```

In the above example, the **validateAge** method can throw a checked exception (Exception), but it does not declare this with the **throws** keyword. The Java compiler will produce an error like this: *Unhandled exception type Exception*.

To fix this, the method must either declare the exception using **throws**, or the statement should be safeguarded with a *try-catch*

block. Here is the corrected code.

```
1. public class BasicThrowsExample {
2.     public static void main(String[] args) {
3.         try {
4.             validateAge(15); // This will throw an exception
5.         } catch (Exception e) {
6.             System.out.println("Error: Invalid
   Age: " + e.getMessage());
7.         }
8.     }
9.
10.    // Declaring that the method throws an Exception
11.    static void validateAge(int age) throws Exception {
12.        if (age < 18) {
13.            // Throwing a checked exception
14.            throw new Exception("Age should be 18 or more");
15.        }
16.        System.out.println("Got the access!");
17.    }
18. }
```

In the corrected example, the **validateAge** method is declared to throw an Exception using the **throws** keyword. The **main** method calls **validateAge** within a *try-catch* block to handle the potential exception. The output will be:

Error: Invalid Age: Age should be 18 or more

finally

In Java, the **finally** clause is used to define a block of code that will always be executed after the **try** block, regardless of whether an exception is thrown or caught. The **finally** block appears at the end of the catch block. The primary purpose of the **finally** block is to ensure that important cleanup operations, such as closing files, releasing resources, or resetting variables, are performed, even if an error occurs.

Here is a simple example demonstrating the use of the **finally** clause:

```
1. public class FinalBlockDemo {
2.     public static void main(String[] args) {
```

```
        try {
            int quotient = calculateQuotient(10, 0); // This will throw an A
            System.out.println("Quotient: " + quotient);
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e.getMessage());
        } finally {
            System.out.println("Executing finally block. This always runs.");
        }
        System.out.println("Program execution continues...");
    }

    static int calculateQuotient(int numerator, int denominator) {
        return numerator / denominator; // This line throws an Arithmeti
    }
}
```

The program will give the following output:

Exception caught: / by zero
Executing finally block. This always runs.
Program execution continues...

The **main** method calls **calculateQuotient(10, 0)**, which throws an **ArithmeticException** because of the attempt of division by zero. The **catch** block catches the **ArithmeticException** and prints an error message. Regardless of the exception, the **finally** block executes, printing **"Executing finally block. This always runs."**. The program then continues with its normal flow, printing **"Program execution continues..."**. The **finally** block is guaranteed to execute, making it the ideal place for cleanup code, ensuring that resources are always properly released.

Built-in checked and unchecked exceptions

Built-in exceptions in Java are part of the Java Exception Handling framework, designed to manage errors and other exceptional events in a standardized manner. These exceptions are predefined by the Java language and kept inside **java.lang** package. This is a standard package of Java, so it is imported to all Java programs implicitly. These classes are broadly classified into two main categories: checked exceptions and unchecked exceptions.

Exceptions that can be predicted by the programmer at the compile time are called checked exceptions. For example, if a required file is not found, this type of exception must be checked at compile time. A method that may throw checked exceptions must declare them in its method signature with the throws keyword, forcing the caller to handle or declare these exceptions. This ensures that these exceptional conditions are anticipated and properly managed. Checked exceptions inherit from the *Throwable* class, which are direct subclasses of *Exception*. Some common checked exceptions are **IOException**, **SQLException**, **ClassNotFoundException**. The checked exceptions are defined inside **java.lang** package are listed in *Table 6.1*.

Let us discuss a program:

```
1. import java.io.File;
2. import java.io.FileReader;
3. import java.io.IOException;
4.
5. public class CustomCheckedExceptionExample {
6.     public static void main(String[] args) {
7.         try {
8.             readCustomFile("nonexistentfile.txt");
9.         } catch (IOException e) {
10.            System.out.println("Encountered an IOException while trying to
11.        }
12.    }
13.
14.    // Declares that it throws IOException, a checked exception
15.    static void readCustomFile(String fileName) throws IOException {
16.        FileReader fileReader = new FileReader(fileName);
17.        // Reading file operations
18.        fileReader.close();
19.    }
20. }
```

In the example above, the program demonstrates handling a checked exception ***(IOException)*** by declaring it with the **throws** keyword, forcing the caller to handle it.

On the other hand, unchecked exceptions are ignored at compile time but checked at runtime. These include subclasses of

RuntimeException, such as **NullPointerException**, **ArrayIndexOutOfBoundsException**, and **IllegalArgumentException**. Unchecked exceptions typically indicate programming errors, such as logic flaws or improper use of an API. They do not need to be declared or caught explicitly, allowing for cleaner code when dealing with common, predictable issues that could be avoided with better coding practices. The unchecked exceptions defined inside **java.lang** package is listed in *Table 6.2*.

The following program demonstrates unchecked exception:

```
1. public class CustomUncheckedExceptionExample {
2.     public static void main(String[] args) {
3.         try {
4.             int result = calculate(10, 0);
5.             System.out.println("Calculation Result: " + result);
6.         } catch (ArithmeticException e) {
7.             System.out.println("Encountered an ArithmeticException: " + e
8.         }
9.     }
10.
11.     // ArithmeticException is an unchecked exception, no need to declar
12.     static int calculate(int number1, int number2) {
13.         return number1 / number2;
14.     }
15. }
```

The above program shows handling an unchecked exception (**ArithmeticException**) without needing to declare it, illustrating the difference in how checked and unchecked exceptions are managed in Java.

Here is the list of checked exceptions which are part of **java.lang** package. These exceptions must be handled using try-catch blocks or declared to be thrown using the throws keyword in method signatures.

Exception	Purpose
IOException	Thrown when an I/O operation fails or is interrupted
SQLException	Thrown when there is an error in database access or other SQL-related issues.
ClassNotFoundException	This exception is thrown when an application tries to load a class through its string name using methods like **Class.forName()**, but the class cannot be

found in the classpath.

InterruptedException	This exception is thrown when a thread that is sleeping, waiting, or otherwise occupied is interrupted.
FileNotFoundException	This exception is thrown when an attempt to open the file denoted by a specified pathname has failed.
EOFException	This exception is thrown when an end of file or end of stream has been reached unexpectedly during input.
InstantiationException	This exception is thrown when an application tries to create an instance of a class using the **newInstance()** method in class **Class**, but the specified class object cannot be instantiated because it is an interface or an abstract class.
IllegalAccessException	This exception is thrown when an application tries to reflectively create an instance, set or get a field, or invoke a method, but the currently executing method does not have access to the definition of the specified class, field, method, or constructor.
NoSuchMethodException	This exception is thrown when a particular method cannot be found.
InvocationTargetException	This exception is thrown to indicate that a method invoked through reflection threw an exception.
MalformedURLException	This exception is thrown when a malformed URL has occurred. It means that the URL could not be parsed.
CloneNotSupportedException	This exception is thrown to indicate that the **clone()** method in class **Object** has been called to clone an object, but that object's class does not implement the Cloneable interface.

***Table 6.1**: List of checked exception classes*

Here is a list of unchecked runtime exception classes in the **java.lang** package:

Exception	Purpose
ArithmeticException	This exception is thrown when an exceptional arithmetic condition has occurred. For example, dividing by zero.
ArrayIndexOutOfBoundsException	This exception is thrown when an array has been accessed with an illegal index.
ArrayStoreException	This exception is thrown to indicate that an attempt has been made to store the wrong type of object into an array of objects.
ClassCastException	This exception is thrown to indicate that the code has attempted to cast an object to a subclass of which it is not an instance.
EnumConstantNotPresentException	This exception is thrown when an application tries to access an enum constant by name and the enum type does not contain a constant with the specified name.
IllegalArgumentException	This exception is thrown to indicate that a method has been passed an illegal or inappropriate argument.
IllegalMonitorStateException	This exception is thrown to indicate that a thread has attempted to wait on an object's monitor or to notify other threads waiting on an object's monitor without owning the specified monitor.
IllegalStateException	This exception is thrown to indicate that a method has been invoked at an illegal or inappropriate time.
IllegalThreadStateException	This exception is thrown when a thread is not in an appropriate state for the

	requested operation.
IndexOutOfBoundsException	Thrown to indicate that an index of some sort (such as to an array, to a string, or to a vector) is out of range.
NegativeArraySizeException	Thrown if an application tries to create an array with negative size.
NullPointerException	Thrown when an application attempts to use null in a case where an object is required.
NumberFormatException	Thrown to indicate that the application has attempted to convert a string to one of the numeric types, but the string does not have the appropriate format.
SecurityException	Thrown by the security manager to indicate a security violation.
StringIndexOutOfBoundsException	Thrown by String methods to indicate that an index is either negative or greater than the size of the string.
TypeNotPresentException	Thrown when an application tries to access a type using a string representing the type's name, but no definition for the type with the specified name can be found.
UnsupportedOperationException	Thrown to indicate that the requested operation is not supported.

***Table 6.2**: List of unchecked exceptions, subclasses of RuntimeException*

These exceptions extend **RuntimeException**, and they represent various runtime errors that can occur during the execution of a program. Since they are unchecked, they do not need to be declared in a method's throws clause or caught.

User-defined exceptions

Developers can create their own exception classes to handle specific error conditions that are not covered by Java's built-in exceptions. This customization enables more meaningful and precise error handling in applications. User-defined exceptions are typically created by extending the **Exception** class for checked exceptions. For unchecked exceptions, the **RuntimeException** class should be extended. This approach enhances code readability and maintainability by providing clear, domain-specific error messages and handling mechanisms.

To create a user-defined exception, you simply define a new class that extends either **Exception** or **RuntimeException**. You can add constructors and methods as needed to provide additional functionality or information. These custom exceptions can then be thrown and caught just like standard exceptions.

The following program demonstrates creating a user-defined exception:

```
// Define a user-defined checked exception
class AgeException extends Exception {
    public AgeException(String message) {
        super(message);
    }
}
```

Let us write another class to use the exception:

```
public class UserDefinedExceptionExample {
    public static void main(String[] args) {
        try {
            isValidAge(13); // This will raise an AgeException
        } catch (AgeException e) {
            System.out.println("Error:Invalid Age: « + e.getMessage());
        }
    }
    // Method that throws the user-defined exception
    static void isValidAge(int age) throws AgeException {
        if (age < 18) {
            throw new AgeException("Age should be 18 or more.");
        }
        System.out.println(age+" is valid age.");
    }
}
```

In the above program, the **AgeException** class extends **Exception**, making it a checked exception. It includes a constructor that takes a message parameter, which is passed to the superclass constructor to set the exception message. In the **UserDefinedExceptionExample** class, the **isValidAge** method checks if the age is less than 18. If so, it throws an **AgeException** with a specific message. The main method calls **isValidAge** within a try-catch block. When **isValidAge** throws an **AgeException**, the catch block catches it and prints the exception message.

Best practices for exception handling in Java

Effective exception handling is crucial for developing robust, maintainable, and easily debuggable Java applications. By following a set of best practices, you can ensure that your application gracefully handles errors and provides useful information to both

developers and users. Below are the best practices for handling exceptions in Java:

Catch specific exceptions, not generic ones

Rather than catching generic exceptions such as Exception or Throwable, you should always catch the most specific exception possible. This allows for precise error handling and avoids unintended behavior. Catching specific exceptions also makes your code more readable and maintainable. For example:

```
try {
    // code that may cause an exception
} catch (FileNotFoundException e) {
    // handle file not found
} catch (IOException e) {
    // handle general I/O issues
}
```

Avoid swallowing exceptions

Swallowing exceptions (i.e., catching exceptions without taking any action) is a bad practice. If you catch an exception, make sure to handle it or at least log it. Swallowed exceptions make it harder to diagnose issues and can leave the application in an unstable state. For example:

```
try {
    // code that may cause an exception
} catch (IOException e) {
    // Log the exception for debugging
    System.err.println("Error: An IOException occurred: " +
e.getMessage());
    e.printStackTrace();
}
```

Use finally for cleanup

Always use the finally block to ensure proper cleanup, such as closing files or releasing database connections. This guarantees that resources are freed even if an exception occurs.

For example:

```
BufferedReader bufferedReader = null;
try {
    bufferedReader = new BufferedReader(new FileReader("file.txt"));
    // read from the file
} catch (IOException e) {
    // Handle IOException
} finally {
    if (bufferedReader != null) {
        try {
            bufferedReader.close();
        } catch (IOException e) {
            // Handle closing error
            e.printStackTrace();
        }
    }
}
```

Log exceptions using a logging framework

Instead of printing stack traces directly to the console, use a logging framework (e.g., SLF4J, Log4J, or java.util.logging) to log exceptions. This provides better control over logging levels (info, warn, error) and makes it easier to track issues in production environments.

For example:

```
private static final Logger logger =
LoggerFactory.getLogger(MyClass.class);
try {
    // code that may cause an exception
} catch (IOException e) {
    logger.error("An IOException occurred", e); // Logs the exception
with the stack trace
}
```

Throw custom exceptions for specific errors

Create and throw custom exceptions to represent specific error conditions in your application. Custom exceptions improve code readability and make error messages more meaningful.

For example:

```
class InvalidUserInputException extends Exception {
    public InvalidUserInputException(String message) {
        super(message);
    }
}
public void processUserInput(String input) throws
InvalidUserInputException {
    if (input == null || input.isEmpty()) {
        throw new InvalidUserInputException("User input cannot be null
or empty");
    }
    // process the input
}
```

Document exceptions with Javadoc

Use Javadoc comments to document the exceptions that a method might throw. This helps other developers understand potential error conditions without needing to dive into the code implementation.

For example:

```
/**
 * Processes the user input.
 *
 * @param input the user input
 * @throws InvalidUserInputException if the input is null or empty
 */
public void processUserInput(String input) throws
InvalidUserInputException {
    // method implementation
}
```

Centralize error codes and descriptions

Instead of hardcoding error messages or error codes throughout your application, centralize them in an error code management system. This could be a set of constants or an enum that stores error codes and their descriptions. Centralizing error codes helps

maintain consistency and makes the code easier to update if error messages or handling procedures change.

For example:

```
public enum ErrorCode {
    INVALID_USER_INPUT("1001", "User input is invalid"),
    FILE_NOT_FOUND("1002", "File not found"),
    DATABASE_ERROR("1003", "Database connection error");
    private final String code;
    private final String description;
    ErrorCode(String code, String description) {
        this.code = code;
        this.description = description;
    }
    public String getCode() {
        return code;
    }
    public String getDescription() {
        return description;
    }
}
public class ApplicationException extends Exception {
    private final ErrorCode errorCode;
    public ApplicationException(ErrorCode errorCode) {
        super(errorCode.getDescription());
        this.errorCode = errorCode;
    }
    public String getErrorCode() {
        return errorCode.getCode();
    }
}
public void processUserInput(String input) throws ApplicationException
{
    if (input == null || input.isEmpty()) {
        throw new
ApplicationException(ErrorCode.INVALID_USER_INPUT);
    }
```

```
    // process the input
}
```

With this approach, if you need to update the error message for a certain error code, you only need to modify it in one place (in the **ErrorCode** enum), rather than in every occurrence throughout your codebase.

Fail fast and loud

When you encounter an error that cannot be handled gracefully, fail fast by throwing an exception immediately. This helps detect issues early rather than letting the application proceed in an unstable state.

For example:

```
public void performOperation(Object obj) {
    if (obj == null) {
        throw new IllegalArgumentException("Null object provided");  // Fail fast
    }
    // Continue operation
}
```

Handle exceptions at the appropriate level

Handle exceptions at the level where you can provide meaningful recovery or useful feedback. Avoid catching exceptions at a high level unless necessary, as it might obscure the root cause of the problem.

For example:

```
try {
    // code that may throw an exception
} catch (IOException e) {
    // Handle I/O error at the appropriate level
    // Provide meaningful feedback or take corrective actions
}
```

Use multi-catch blocks for related exceptions

From Java 7 onward, you can use the multi-catch block to catch

multiple related exceptions in a single catch block. This makes your code more concise and improves readability.

For example:

```
try {
    // code that may throw multiple exceptions
} catch (IOException | SQLException e) {
    // Handle both IOException and SQLException
    System.err.println("Error occurred: " + e.getMessage());
}
```

Following these best practices will help you write clean, efficient, and maintainable Java applications. By handling exceptions properly, logging them effectively, and using centralization techniques for error management, you can create more robust applications that are easier to debug and extend.

Conclusion

In this chapter, we explored the exceptions and their handling in Java program. The Exception class and its hierarchy form the foundation of Java's error-handling mechanism, encompassing both checked and unchecked exceptions. Key constructs like try, catch, throw, throws, and finally enable developers to handle and manage exceptions effectively. Standard exceptions encompass typical error scenarios, while user-defined exceptions allow for application-specific error handling. Adhering to effective exception handling techniques, such as using specific exceptions, ensuring resource cleanup, and logging errors, enhances the reliability and readability of your code. Mastering these concepts ensures your applications can gracefully handle unexpected conditions and maintain consistent performance.

In the next chapter, we will learn about input-output operations in Java programming.

Exercises

1. Write a Java program that initializes a string to null and then attempts to call a method on it. Catch the NullPointerException and print an appropriate message.

2. Write a Java program that initializes an array with five elements and then attempts to access an index that is out of bounds. Catch the ArrayIndexOutOfBoundsException and print an appropriate message.
3. Write a Java program that performs the division of two integers where the denominator is zero. Catch the ArithmeticException and print an appropriate message.
4. Create a custom exception named NegativeNumberException. Write a method that takes an integer and throws this exception if the integer is negative. In the main method, call this method with a negative number and handle the custom exception.
5. Write a Java program that contains a method to calculate the square root of a number. If the number is negative, the method should throw an IllegalArgumentException. In the main method, call this method with a negative number and handle the exception.
6. Write a Java program that attempts to convert a string containing non-numeric characters to an integer using Integer.parseInt. Catch the NumberFormatException and print an appropriate message.
7. Write a Java program that performs division and array access. Write multiple catch blocks to handle ArithmeticException and ArrayIndexOutOfBoundsException separately.
8. Create a custom exception named InsufficientFundsException. Write a class to implement a bank account with methods for deposit, withdrawal, and showing balance. The withdraw method should throw InsufficientFundsException if the withdrawal amount is greater than the account balance. Test this class in your main method by trying to withdraw more money than available and catching the custom exception.

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the

Authors:

https://discord.bpbonline.com

Chapter 7

Input/Output Operations

Introduction

This chapter introduces readers to **input/output (I/O)** operations in Java, enabling interaction with external data sources and storage. Understanding I/O operations is essential for building applications that read from and write to files, streams, and other data sources. I/O operations form a critical part of any programming language, as they allow programs to interact with the external world. In Java, the I/O system is both powerful and versatile, enabling developers to perform a wide range of operations, from reading user input from the console to processing large files and communicating over networks. This chapter delves into the intricacies of Java's I/O mechanisms, exploring the key concepts, classes, and practical examples that will help you to understand and effectively handle I/O operations in your Java applications.

Structure

The chapter covers the following topics:

- Elementary concepts of input/output
- Streams
- Standard input/output
- File input/output
- Automatic file closing

Objectives

The objective of this chapter is to provide a comprehensive understanding of Java's I/O capabilities, focusing on essential concepts and practical applications. By the end of this chapter, readers will grasp the fundamental principles of I/O operations in Java, including the significance of streams and efficient data handling. They will learn to differentiate between byte streams and character streams, understanding when and how to use each type. The chapter will help readers learn the skills to perform various file I/O operations using the **java.io** package, including creating, reading, writing, and deleting files and directories. Additionally, readers will understand the importance of resource management in I/O operations and learn to implement automatic file closing using try-with-resources statements. This foundation will enable readers to handle a wide range of I/O tasks, from simple console input and output to complex file and network communications, preparing them for more advanced Java I/O topics and applications.

Elementary concepts of input/output

In our preceding chapters, we have seen very little about input/output operations in example programs. This is because, in real life applications, Java mostly uses a graphical interface for interacting with users. Java is rarely used for writing text-based console applications. Although the text-based applications are very good for learning, they have less significance in real world Java development. Java’s support for console I/O is limited and often requires a significant amount of boilerplate code. For example, setting up streams, handling exceptions, and ensuring resources are closed properly can make simple tasks cumbersome.

Java provides two packages **java.io** and **java.nio**, for input/output

support. These operations in Java are API based. Java does not provide any keywords for performing I/O operations. The **java.io** package has been available since Java's early versions and provides classes for system input and output through data streams, serialization, and file system operations. It follows a stream-oriented (sequential) I/O model, which is straightforward but can be slower for large-scale or highly concurrent I/O operations. The **java.nio** (New I/O) package was introduced in Java 1.4 to provide an alternative to the traditional I/O with more efficient, scalable, and flexible I/O operations. It follows a buffer-oriented and non-blocking I/O model, which is better suited for high-performance I/O operations, especially in network programming and large-scale applications. In this chapter, we will focus on stream-based I/O.

Streams

I/O operations in Java are performed using streams. Streams are like channels through which data flows. Imagine you have a pipeline that transports water from one place to another. Similarly, streams in Java transport data from one place to another. A stream is a sequence of data elements made available over time, providing a way to read from or write to a data source or destination. Streams abstract the details of these data sources and sinks, allowing developers to work with data in a consistent and uniform manner. Streams in Java are linked to a physical device and provide a unified approach to handling I/O operations, regardless of the underlying physical device. By abstracting the details of data sources and destinations, streams allow developers to interact with various I/O devices, such as files, network sockets, and keyboard inputs, using a consistent API. This means the same basic stream operations—like reading, writing, buffering, and closing—can be applied to different types of I/O sources without needing to understand the specifics of each device. For instance, whether you are reading data from a file on disk, receiving data over a network, or capturing user input from the console, you use the same **InputStream** or **Reader** classes for input and the **OutputStream** or **Writer** classes for output. This unification simplifies the process of managing I/O operations in Java, enabling code reusability and reducing complexity, as developers can focus on handling data

streams rather than dealing with the intricacies of each physical device.

InputStream is used to read data from a source. Think of this as a pipe through which data flows into your program. Example: Reading from a file or getting user input from the keyboard. OutputStream isused to write data to a destination. This is like a pipe through which data flows out of your program. Example: Writing data to a file or displaying output on the screen.

Types of streams

Java's I/O system categorizes streams into two main types: Byte streams and character streams. Byte streams handle the I/O of raw binary data. They are used to read and write data in binary format (8-bit bytes). The primary classes for byte streams are **InputStream** and **OutputStream**, which have various subclasses for different data sources and sinks. All these classes are part of the **java.io** package.

Character streams handle I/O of character data (16-bit Unicode characters). They are used to read and write text files and handle character encoding properly. The primary classes for character streams are **Reader** and **Writer**, which have various subclasses for different data sources and sinks. All these classes are kept inside **java.io** package, so for using any of these classes we need to import the class into the program.

Byte stream classes

The byte stream classes in Java are defined as a structured hierarchy to provide a versatile framework for handling binary I/O operations. At the top of this hierarchy are the abstract classes **InputStream** and **OutputStream**, which define the basic methods for reading and writing bytes, respectively. **InputStream** is an abstract class representing an input stream of bytes, and it defines several essential methods. The most important method is **read()**. Similarly, **OutputStream** is an abstract class that represents an output stream of bytes. Its main method is **write(int b***)*, which writes the specified byte to the output stream. All byte stream classes of **java.io** package, which implement the reading process from any source derive from **InputStream** class, and classes which

implement the writing/printing process to any destination derived from **OutputStream** class. In order to use these classes in a program, you need to import **java.io**package.

The byte stream classes are listed in *Table 7.1:*

Class	Description
InputStream	The superclass for all byte input streams. Defines methods for reading bytes from a source.
FileInputStream	Reads bytes from a file.
ByteArrayInputStream	Reads bytes from a byte array.
FilterInputStream	A base class that filters data read from an underlying input stream.
BufferedInputStream	Provides efficient reading of bytes.
DataInputStream	Allows an application to read primitive data types (e.g., int, float) from an underlying input stream.
ObjectInputStream	Deserializes primitive data and objects previously written using an **ObjectOutputStream**.
PipedInputStream	Represents a communication pipe between two threads.
SequenceInputStream	Combines multiple input streams into a single input stream, reading them sequentially.
OutputStream	The superclass for all byte output streams. Defines methods for writing bytes to a destination.
FileOutputStream	Writes bytes to a file.
ByteArrayOutputStream	Writes bytes to a byte array.
FilterOutputStream	A base class that filters data written to an underlying output stream.
BufferedOutputStream	Provides efficient writing of bytes.
DataOutputStream	Allows an application to write primitive data types (e.g., int, float) to an underlying output stream.
ObjectOutputStream	Serializes primitive data and objects to an output stream.
PipedOutputStream	Represents a communication pipe between two threads.

Table 7.1*: Byte stream class and their purpose*

Common byte stream classes and their methods

In Java, byte stream classes are used for input and output of 8-bit bytes. These classes are part of the **java.io** package and include a variety of classes for reading from and writing to files, arrays, and other sources. Here are some common byte stream classes along with their primary methods.

InputStream

InputStream is an abstract class that represents an input stream of bytes. It provides a standard interface for reading bytes from a

source. The **InputStream** class is the superclass of all classes representing an input stream of bytes. The purpose of the class is to read raw bytes. The commonly used methods of **InputStream** class are listed as follows:

Method	Description
int read()	Reads the next byte of data from the input stream. Returns **-1** if the end of the stream is reached.
int read(byte[] b)	Reads up to **b.length** bytes of data into an array of bytes and returns the number of bytes read.
int read(byte[] b, int off, int len)	Reads up to **len** bytes of data from the input stream into an array of bytes, starting at the specified offset.
void reset()	Resets the input stream to the beginning.
void close()	Closes the input stream and releases any system resources associated with it.

***Table 7.2**: Commonly used methods of InputStream class and their purpose*

Commonly used subclasses of **InputStream** include **FileInputStream** for reading bytes from a file, **ByteArrayInputStream** for reading from a byte array, and **BufferedInputStream** for reading bytes from an underlying input stream with buffering to improve efficiency.

OutputStream

OutputStream is an abstract class that represents an output stream of bytes. It provides a standard interface for writing bytes to a destination. The **OutputStream** class is the superclass of all classes representing an output stream of bytes. Purpose of the class is to write raw byte data.

Commonly used methods of **OutputStream** class are as follows:

Method	Description
void write(int b)	Writes the specified byte to the output stream.
void write(byte[] b)	Writes all bytes from the specified byte array to the output stream.
void write(byte[] b, int off, int len)	Writes **len** bytes from the specified byte array starting at off to the output stream.
flush()	Flushes the stream and forces any buffered output bytes to be written out.
void close()	Closes the output stream and releases any system resources associated with it.

***Table 7.3**: Commonly used methods of OutputStream class and their purpose*

Primary subclasses of **OutputStream** include **FileOutputStream** for writing bytes to a file, **ByteArrayOutputStream** for writing to a byte array, and **BufferedOutputStream** for writing bytes to an underlying output stream with buffering.

DataOutputStream

DataOutputStream is a filter output stream that allows an application to write primitive Java data types to an output stream in a machine-independent way. This is part of the *java.io* package, and its base class is **FilterOutputStream**. **FilterOutputStream** is a common superclass for all classes that filter output streams. These streams sit on top of an already existing output stream (the underlying output stream), which it uses as its basic sink of data, but it could possibly transform the data along the way or provide additional functionality. In this hierarchy, **DataOutputStream** extends **FilterOutputStream**, which extends **OutputStream**. This design allows **DataOutputStream** to inherit methods from **FilterOutputStream** and **OutputStream** and to override or extend their functionality. The purpose of this class is to write primitive data types to an output stream.

Constructors:

DataOutputStream(OutputStream out)

Creates a **DataOutputStream** that uses the specified underlying **OutputStream**.

Common methods:

Method	Description
void writeBoolean(boolean v)	Writes a boolean value to the output stream.
void writeByte(int v)	Writes a byte value to the output stream.
void writeChar(int v)	Writes a char value to the output stream.
void writeDouble(double v)	Writes a double value to the output stream.
void writeFloat(float v)	Writes a float value to the output stream.
void writeInt(int v)	Writes an int value to the output stream.
void writeLong(long v)	Writes a long value to the output stream.
void writeShort(int v)	Writes a short value to the output stream.
void writeUTF(String s)	Writes a string in UTF-8 format to the output stream.

***Table 7.4**: Commonly used methods of DataOutputStream class and their purpose*

DataInputStream

DataInputStream is a filter input stream that allows an application to read primitive Java data types from an underlying input stream in a machine-independent way. This is part of the **java.io** package and its base class is **FilterInputStream**, which extends **InputStream**. This design allows **DataInputStream** to inherit methods from **FilterInputStream** and **InputStream**, and to override or extend their functionality.

Constructors:

DataInputStream(InputStream in)

It creates a **DataInputStream** that uses the specified underlying **InputStream**.

Common methods:

Method	Description
boolean readBoolean()	Reads a boolean from the input stream.
byte readByte()	Reads a byte from the input stream.
char readChar()	Reads a character from the input stream.
double readDouble()	Reads a double from the input stream.
float readFloat()	Reads a float from the input stream.
void readFully(byte[] b)	Reads bytes from the input stream.
int readInt()	Reads an integer from the input stream.
long readLong()	Reads a long from the input stream.
short readShort()	Reads a short from the input stream.
String readUTF()	Reads a string in modified UTF-8 format from the input stream.

***Table 7.5**: Commonly used methods of DataInputStream class and their purpose*

Character stream classes

The character stream classes in Java provide a framework for handling I/O operations involving character data, ensuring proper handling of character encoding and decoding. At the top of this hierarchy are the abstract classes **Reader** and **Writer**, which define the basic methods for reading and writing characters, respectively. The character stream classes are listed in *Table 7.6*:

Class	Description
Reader	An abstract superclass for all character input streams. Defines methods for reading characters from a source.
FileReader	Reads characters from a file.
CharArrayReader	Reads characters from a character array.

StringReader	Reads characters from a string.
BufferedReader	Provide efficient reading of characters, arrays, and lines.
InputStreamReader	Bridges byte streams to character streams by decoding bytes into characters using a specified charset.
FilterReader	Abstract class for reading filtered character streams.
PipedReader	Connects to a piped writer and reads characters from it.
LineNumberReader	A buffered character input stream that keeps track of line numbers.
PushbackReader	A character stream reader that allows characters to be pushed back into the stream.
Writer	Abstract superclass for all character output streams. Defines methods for writing characters to a destination.
FileWriter	Writes characters to a file.
CharArrayWriter	Writes characters to a character array.
StringWriter	Writes characters to a string buffer, which can be used to construct a string.
BufferedWriter	Provide efficient writing of characters, arrays, and lines.
OutputStreamWriter	Bridges character streams to byte streams by encoding characters into bytes using a specified charset.
FilterWriter	Abstract class for writing filtered character streams.
PipedWriter	Connects to a piped reader and writes characters to it.
PrintWriter	Provides convenient methods for printing formatted representations of objects to a text-output stream.

***Table 7.6**: Character stream class and their purpose*

Common character stream classes and their methods

In Java, character stream classes are used for input and output of 16-bit Unicode characters. These classes are part of the **java.io** package and include a variety of classes for reading from and writing to files, arrays, and other sources. Here are some common character stream classes along with their primary methods.

Reader

The Reader class in Java is an abstract class that represents the character stream input. It is part of the **java.io** package and serves as the base class for all classes that read character streams. Unlike byte streams, which handle raw binary data, character streams handle text data, allowing for easier manipulation of characters and strings. **Reader** is an abstract class, meaning it cannot be instantiated directly. Instead, you use its concrete subclasses, such as **FileReader**, **BufferedReader**, **InputStreamReader**, etc.

Commonly used methods:

Method	Description
int read()	Reads a single character. Returns the character read, or **-1** if the end of the stream has been reached.
int read(char[] cbuf)	Reads characters into an array. Returns the number of characters read, or **-1** if the end of the stream has been reached.
int read(char[] cbuf, int off, int len)	Reads characters into a portion of an array. Returns the number of characters read, or **-1** if the end of the stream has been reached.
void mark(int readAheadLimit)	Marks the present position in the stream. Subsequent calls to **reset()** will reposition the stream to this point.
void reset()	Resets the stream to the most recent mark.
void close()	Closes the stream and releases any system resources associated with it.

***Table 7.7**: Commonly used methods of Reader class and their purpose*

InputStreamReader

The **InputStreamReader** class in Java is a bridge from byte streams to character streams. It reads bytes and decodes them into characters using a specified charset. If no charset is specified, it uses the platform's default charset. It is part of the **java.io** package and is essential when you need to read text data from a byte stream. It is useful for reading text data from various sources like files, network connections, etc., especially when the text is in a specific encoding.

Constructors:

InputStreamReader(InputStream in)

Creates an **InputStreamReader** that uses the default charset.

Example:

```
InputStream inputStream = new FileInputStream("example.txt");
InputStreamReader reader = new InputStreamReader(inputStream);
```

InputStreamReader(InputStream in, String charsetName)

Creates an **InputStreamReader** that uses the specified charset.

Example:

```
InputStream inputStream = new FileInputStream("example.txt");
InputStreamReader reader = new InputStreamReader(inputStream, "UTF-8");
```

Common methods:

Since **InputStreamReader** extends **Reader**, it inherits all the

methods from the **Reader** class.

FileReader

The **FileReader** class in Java is a convenience class for reading character files. It extends the **InputStreamReader** class, making it possible to read files as a stream of characters, thus simplifying the process of file input for text data.

Constructors:

FileReader(String fileName)

Creates a new **FileReader**, given the name of the file to read from.

Example:

```
FileReader reader = new FileReader("example.txt");
```

FileReader(File file)

Creates a new **FileReader**, given the **File** object to read from.

Example:

```
File file = new File("example.txt");
FileReader reader = new FileReader(file);
```

Common methods:

Since **FileReader** extends **InputStreamReader**, it inherits all its methods.

BufferedReader

The **BufferedReader** class in Java is a character input stream that provides buffering to the *Reader* class. It is used to read text from an input stream efficiently by reducing the number of I/O operations. It uses a buffer to read characters, arrays, or lines, which improves performance by minimizing the number of read operations from the underlying stream. This class provides methods to read text line by line, making it easier to process **.text** files. It is considered most suitable for reading large files or data streams where frequent read operations are required.

Constructors:

BufferedReader(Reader in)

Creates a buffered character input stream using the default buffer size.

Example:

BufferedReader reader = new BufferedReader(new FileReader("example.txt"));

BufferedReader(Reader in, int sz)

Creates a buffered character input stream using the specified buffer size.

Example:

BufferedReader reader = new BufferedReader(new FileReader("example.txt"), 8192);

Common methods:

String readLine(): Reads a line of text. A line is considered to be terminated by any one of a line feed (**n**), a carriage return (**r**), or a carriage return followed immediately by a line feed. Returns a String containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached.

Example:

```
BufferedReader reader = new BufferedReader(new
FileReader("example.txt"));
String line = reader.readLine();
while (line != null) {
   System.out.println(line);
   line = reader.readLine();
}
reader.close();
```

int read(): Reads a single character. Returns the character read as an integer in the range 0 to 65535 (0x00-0xffff), or -1 if the end of the stream has been reached.

Writer

The **Writer** class in Java is an abstract class that represents character stream output. It serves as the base class for all classes that write character streams. This class is the counterpart to the **Reader** class and is used for writing characters, arrays of characters, and strings. **Writer** is an abstract class, so it cannot be instantiated directly. Instead, you use its concrete subclasses, such as

FileWriter, **BufferedWriter**, **PrintWriter**, etc. Designed to write streams of characters, making it suitable for text data. It can handle different character encodings when used with subclasses like **OutputStreamWriter**.

Commonly used methods:

Method	Description
Writer.write(int c)	Writes a single character.
Writer.write(char[] cbuf)	Writes an array of characters.
Writer.write(char[] cbuf, int off, int len)	Writes a portion of an array of characters.
Writer.write(String str)	Writes a string.
Writer.write(String str, int off, int len)	Writes a portion of a string.
Writer.flush()	Flushes the stream, ensuring that any buffered characters are written immediately.
Writer.close()	Closes the stream and releases any system resources associated with it.

***Table 7.8**: Commonly used methods of Writer class and their purpose*

FileWriter

The **FileWriter** class in Java is a subclass of **Writer** that simplifies the task of writing characters to files. It is specifically designed for writing text data to files.

Constructors:

FileWriter(String fileName)

Creates a **FileWriter** using the given file name.

Example:

```
FileWriter writer = new FileWriter("example.txt");
```

FileWriter(String fileName, boolean append)

Creates a **FileWriter** using the given file name and a Boolean indicating whether to append data to the end of the file.

Example:

```
FileWriter writer = new FileWriter("example.txt", true); // Appends to the file
```

FileWriter(File file)

Creates a **FileWriter** using the given **File** object.

```
File file = new File("example.txt");
FileWriter writer = new FileWriter(file);
```

FileWriter(File file, boolean append)

Creates a **FileWriter** using the given **File** object and a Boolean indicating whether to append data to the end of the file.

Example:

```
File file = new File("example.txt");
FileWriter writer = new FileWriter(file, true); // Appends to the file
```

Methods of **FileWriter** are inherited from **Writer** since it extends from it.

PrintWriter

The **PrintWriter** class in Java is a character stream class that provides methods to print formatted representations of objects to a text-output stream. It is often used for producing human-readable output. It offers convenient methods of **print()** and **println()** for various data types. It also provides methods like **printf()** and **format()** for formatted output. It uses an internal buffer to enhance performance. Unlike others, it does not throw *IOException* directly; instead, it uses the **checkError()** method to detect errors. **PrintWriter** class provides auto-flush feature, can be configured to automatically flush the buffer on certain conditions (e.g., when a newline is written).

This class provides many constructors. Some are shown here.

PrintWriter(Writer out)

Creates a new **PrintWriter**, without automatic line flushing.

Example:

```
PrintWriter writer = new PrintWriter(new FileWriter("example.txt"));
```

PrintWriter(Writer out, boolean autoFlush)

Creates a new **PrintWriter** with optional automatic line flushing.

Example:

```
PrintWriter writer = new PrintWriter(new FileWriter("example.txt"), true);
```

PrintWriter(OutputStream out)

Creates a new **PrintWriter** from an existing **OutputStream** without automatic line flushing.

Example:

PrintWriter writer = new PrintWriter(new FileOutputStream("example.txt"));

PrintWriter(OutputStream out, boolean autoFlush)

Creates a new **PrintWriter** from an existing **OutputStream**, with optional automatic line flushing.

Example:

PrintWriter writer = new PrintWriter(new FileOutputStream("example.txt"), true);

PrintWriter(String fileName):

Creates a new **PrintWriter** with the specified file name without automatic line flushing.

Example:

PrintWriter writer = new PrintWriter("example.txt");

Common methods:

The **PrintWriter** class inherits from **Writer** and includes additional methods for printing various data types and formatting output:

	Description	
Void print(boolean b)		
Void print(int i)		
Void print(long l)		
Void print(float f)		
Void print(double d)		
Void print(char[] s)		
Void print(String s)		
Void print(Object obj)	Prints an object by calling its **toString()** method.	
Void println()	Prints a line separator.	
Void println(boolean b)	Prints a boolean value followed by a line separator.	
Void println(int i)	Prints an integer followed by a line separator.	
Void println(long l)	Prints a long integer followed by a line separator.	
Void println(float f)	Prints a float followed by a line separator.	
Void println(double d)	Prints a double followed by a line separator.	
Void println(char[] s)	Prints a character array followed by a line separator.	
Void println(String s)	Prints a string followed by a line separator.	
Void println(Object obj)	Prints an object followed by a line separator.	
PrintWriter printf(String format, Object... args)	Prints a formatted string using the specified format string and arguments.	
PrintWriter format(String format, Object... args)	Prints a formatted string using the specified format string and arguments.	

***Table 7.9**: Commonly used methods of PrintWriter class and their purpose*

Following diagram clarify the relationships and hierarchy within the Java I/O stream classes.

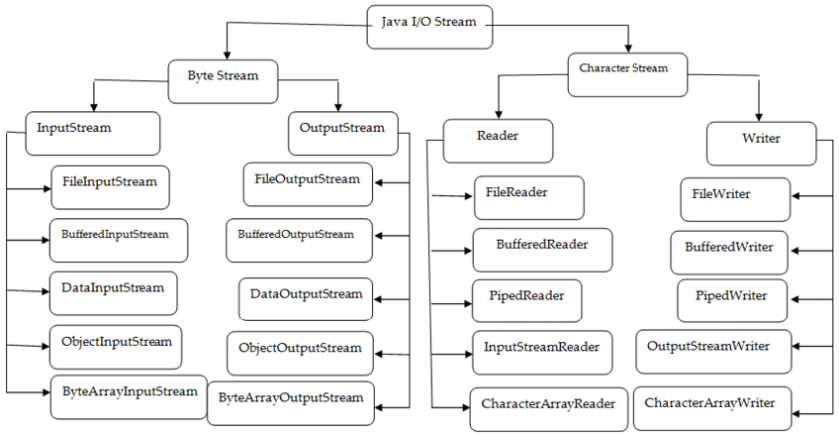


***Figure 7.1**: Java I/O stream class hierarchy*

Standard input/output

The Java runtime environment provides some predefined streams for common input and output tasks. These streams are automatically initialized and made available to every Java application, facilitating the basic I/O operations without the need for additional setup.

For this purpose, Java provides the **System** class that belongs to the **java.lang** package. It is a final class, so it cannot be subclassed. It is designed to provide access to system-related resources and utilities. This class contains several useful class fields and methods that enable interaction with the standard input, output, and error streams, as well as methods for accessing system properties, environment variables, and other system-related operations. The **System** class is automatically available in all Java programs without the need to import it explicitly because it is part of the core Java library. This class contains three predefined streams, **in**, **out** and **err**. These variables are defined as **public** and **static**. So, they can be used anywhere in the program by using the class name. The predefined streams are as follows:

- **System.in**: The standard input stream, an instance of **InputStream**, typically used to read input from the keyboard.

- **System.out**: The standard output stream, an instance of **PrintStream**, used to print output to the console.
- **System.err**: The standard error stream, also an instance of **PrintStream** is used to print error messages to the console.

We have already used **System.out** in all our programs in previous chapters to print on output stream which is console by default. Same way we can use **System.err** for printing error messages on error stream, which is also console. Using **System.in** requires some additional steps, which we are going to learn in the coming section.

Reading data from user

The process of reading data provided by the user through the console or terminal is called as console input. This is typically achieved using the **System.in** stream, which is the standard input stream in Java. However, **System.in** is an instance of **InputStream**, which reads raw byte data, so it is often used in conjunction with other classes like **Scanner** or **BufferedReader** to read and interpret character data more easily.

Reading bytes directly

You can read bytes directly from **System.in** using the **read()** method of **InputStream**. This method reads one byte of data at a time and returns it as an **int** value. The **read()** method blocks until input data is available, an I/O error occurs, or the end of the stream is reached. If the end of the stream is reached, it returns **-1**.

```
1. import java.io.IOException;
2. import java.io.InputStream;
3.
4. public class ByteInputExample {
5.     public static void main(String[] args) {
6.         InputStream inputStream = System.in;
7.         System.out.println("Enter some text:");
8.         try {
9.             int byteData;
10.            while ((byteData = inputStream.read()) != -1) {
11.                System.out.print((char) byteData);
12.            }
13.        } catch (IOException e) {
```

```
14.             System.err.println("An error occurred during I/O operations.");        }
15.     }
16. }
```

Here is the output:

Enter some text:
Hello, World!
Hello, World!

In this example, each byte read is cast to a **char** to display it as a character. However, reading from console using byte stream is not recommended. It is not efficient for reading large amounts of data because it reads one byte at a time.

Rather console input using character stream is preferred and more common in Java applications, you generally wrap **System.in** with a **Reader** class, such as **InputStreamReader**, and often with **BufferedReader** to read entire lines of text efficiently.

Reading characters using InputStreamReader

Let us see one example program to read using **InputStreamReader** class. **InputStreamReaderconverts** byte streams to character streams. The **InputStreamReader** class provide a constructor **InputStreamReader(InputStream in)**. It takes an **InputStream**. To get and **InputStreamReader** linked to **System.in** we can use this constructor as **System.in** refers to an **InputStream** object. and use it as a source of bytes to convert into characters. This creates a new instance of **InputStreamReader** that wraps around **System.in**.

InputStreamReader inputStreamReader = new InputStreamReader(System.in);

Here is an example of how to use it:

```
1. import java.io.IOException;
2. import java.io.InputStreamReader;
3.
4. public class CharInputExample {
5.     public static void main(String[] args) {
6.         InputStreamReader inputStreamReader = new InputStreamReade
7.         System.out.println("Enter some text:");
8.         try {
```

```
            int charData;
            while ((charData = inputStreamReader.read()) != -1) {
                System.out.print((char) charData);
            }
        } catch (IOException e) {
            System.err.println("An error occurred during I/O operations.");
        }
    }
}
```

The program will prompt the user to enter some text. When the user types **Hello, World!** followed by pressing *Enter* and then *Ctrl+D (EOF)* to end the input stream, the output will be as follows:

Enter some text:
Hello, World!
Hello, World!

Similar to the byte input example, the program reads each character from *System.in* and prints it until the end of the stream (EOF) is reached.

InputStreamReader reads data from **System.in** one character at a time. It directly reads and converts bytes into characters. Suitable for simple, low-level character input. This implementation is less efficient for reading large amounts of data because it reads one character at a time.

Reading using BufferedReader

We can perform console input more efficiently by using **BufferedReader** class. **BufferedReader** can wrap an **InputStreamReader** to buffer the input and read lines of text efficiently. One of the constructor of **BufferedReader** is **BufferedReader(Reader reader)**. **Reader** is an abstract class. **InputStreamReader** is one of its subclasses. We can get **InputStreamReader** linked to **System.in**, as we saw in the above section which can be passed as an argument to **BufferedReader** constructor to get a buffered stream on **System.in**. Putting it all together to create a **BufferedReader** linked to the keyboard, the statement will be:

BufferedReader bin = new BufferedReader(new

InputStreamReader(System.in));

Here is how you can use it:

```
1. import java.io.BufferedReader;
2. import java.io.IOException;
3. import java.io.InputStreamReader;
4.
5. public class BufferedReaderExample {
6.     public static void main(String[] args) {
7.         BufferedReader bufferedReader = new BufferedReader(new Input
8.         System.out.println("Enter some text:");
9.         try {
10.             String line;
11.             while ((line = bufferedReader.readLine()) != null) {
12.                 System.out.println(line);
13.             }
14.         } catch (IOException e) {
15.                 System.err.println("An error occurred during I/O
    operations.");
16.         }
17.     }
18. }
```

When the user types **Hello, World!** and then presses *Enter* followed by *Ctrl+D (EOF)* to end the input stream. The output of the program will remain same as previous example. But use of **BufferedReader** simplifies code with methods like **readLine()** and is more efficient due to its internal buffering. This method of console input is ideal for reading lines of text or large amounts of data efficiently.

The program reads an entire line from **System.in** and prints it until the end of the stream (EOF) is reached.

For all these examples, pressing *Ctrl+D* (EOF) signifies the end of input in a terminal (on Unix-like systems such as Linux and macOS). On Windows, the equivalent is *Ctrl+Z* followed by Enter. The actual method to simulate EOF might vary depending on the terminal or command prompt being used.

Reading using Scanner Class

One of the simplest and most preferred method of console input is

by using **Scanner** class.

This class is a part of **java.util** package.(Although it is not a part of **java.io** package unlike others, it finds a mention here because it is it easy to use for beginners and convenient for quick input tasks.) It provides methods for reading various types of input, such as strings, integers, and other primitive types, making it versatile for different input operations. The **Scanner** class can be used to read input from different sources like the console (standard input), files, and strings. This class provides various constructors. A few of them are listed below:

Scanner(InputStream source) **// Reads from an input stream, typically System.in**
Scanner(File source) **// Reads from a file**
Scanner(String source) **// Reads from a string**

The following are the commonly used methods:

Method	Description
Reads the next integer	
Reads the next double	
Reads the next boolean	
Reads the next line of text	
Reads the next token (word)	

***Table 7.10**: Commonly used methods of java.util.Scanner class and their purpose*

Let us see one example program to read using a scanner class:

Example: Reading different types of data

```
1. import java.util.Scanner;
2.
3. public class UserInputExample {
4.     public static void main(String[] args) {
5.         Scanner scanner = new Scanner(System.in);
6.
7.         // Reading a string input
8.         System.out.print("Enter your name: ");
9.         String name = scanner.nextLine();
10.
11.        // Reading an integer input
12.        System.out.print("Enter your age: ");
```

```
        int age = scanner.nextInt();

        // Reading a double input
        System.out.print("Enter your height in meters: ");
        double height = scanner.nextDouble();

        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Height: " + height);

        scanner.close();
    }
}
```

Printing on console

Printing on the console in Java is primarily done using the **System.out** object, which is an instance of **PrintStream**. Let us consider an example that demonstrates various ways to print to the console in Java. This example will show how to use **print()**, **println()**, and **printf()** to display different types of data:

```
public class ConsolePrintingExample {
    public static void main(String[] args) {
        // Using print() to print without a newline
        System.out.print("Hello, ");
        System.out.print("world!");

        // Move to the next line using println()
        System.out.println();

        // Using println() to print with a newline
        System.out.println("This is a new line.");

        // Using printf() for formatted printing
        String name = "Alice";
        int age = 30;
        double height = 5.7;
        //using println to print string and variable
        System.out.println("Name: " + name);

```

```
20.
21.         System.out.printf("Name: %s, Age: %d, Height: %.1f feet
    %n", name, age, height);
22.
23.         // Using println() to print multiple lines
24.         System.out.println("Here are some numbers:");
25.         System.out.println(1);
26.         System.out.println(2);
27.         System.out.println(3);
28.
29.         // Using print() to print numbers on the same line
30.         System.out.print(4);
31.         System.out.print(", ");
32.         System.out.print(5);
33.         System.out.print(", ");
34.         System.out.print(6);
35.         System.out.println();
36.
37.         // Using printf() for alignment
38.         System.out.printf("%-10s %-10s %-10s
    %n", "Name", "Age", "Height");
39.         System.out.printf("%-10s %-10d %-10.1f
    %n", "Bob", 25, 5.9);
40.         System.out.printf("%-10s %-10d %-10.1f
    %n", "Carol", 32, 5.4);
41.     }
42. }
```

The provided Java program demonstrates various methods of printing to the console using the **System.out** object. Initially, it shows the use of **print()** to print text without a newline, followed by **println()** to add a newline, effectively moving the cursor to the next line. The first **println()**, without any arguments, moves the cursor to the next line. The next **println()** is used to print text followed by a newline character. Next **println()** demonstrates how to print messages along with the values of variables.

It then introduces **printf()** for formatted output, demonstrating how to include variables in a formatted string with specific format specifiers. The program further illustrates the use of **println()** for printing multiple lines of numbers and **print()** for printing numbers on the same line, separated by commas. Lastly, it showcases the use

of **printf()** to print aligned columns of data, ensuring the text is neatly organized in a tabular format.

The output will be:

Hello, world!
This is a new line.
Name: Alice
Name: Alice, Age: 30, Height: 5.7 feet
Here are some numbers:
1
2
3
4, 5, 6
Name Age Height
Bob 25 5.9
Carol 32 5.4

Handling error messages in Java

In Java, error messages are typically printed to the standard error stream (**System.err**). This stream is similar to the standard output stream (**System.out**), but it is used specifically for error messages and diagnostics. By printing error messages to **System.err**, you can separate them from normal program output, which can be helpful for debugging and logging. Following code demonstrates the difference between a normal message and an error message.

```
// Simulate a normal message
System.out.println("Program started successfully.");
// Simulate an error message
System.err.println("Error: Unable to connect to the database.");
```

This line uses **System.err.println()** to print an error message to the console. This message will be directed to the standard error stream, which can be redirected separately from the standard output stream if needed.

File input /output

File input and output in Java involves reading from and writing to files on the disk. Java provides a comprehensive set of classes in the

java.io package to handle file operations. The most commonly used classes for file I/O are **FileReader**, **FileWriter**, **BufferedReader**, **BufferedWriter**, **PrintWriter**, **FileInputStream**, and **FileOutputStream**.

File input/output using byte stream

Byte streams in Java are used for handling raw binary data. They operate directly on bytes and are generally used for file I/O operations, especially when dealing with binary data like images, audio, or any other non-text data. Key classes are **FileInputStream**, which reads raw byte data from a file and **FileOutputStreamto** write raw byte data to a file.

Reading from file using byte stream

To read data from a file using byte streams, **FileInputStreamis** the class, which is a subclass of **InputStream**. To open a file, we just need to create an object of this class, specifying the name of the file as an argument to the constructor. This class provides many constructors as we saw in the previous section. We can use any of them. For example, we will be using:

public FileInputStream(String name) throws FileNotFoundException.

This constructor is used to create an input stream linked to the file specified with the given filename. If the file does not exist, it throws **FileNotFoundException**.

Here is a detailed example:

```
1. import java.io.FileInputStream;
2. import java.io.FileNotFoundException;
3. import java.io.IOException;
4.
5. public class FileInputStreamExample {
6.     public static void main(String[] args) {
7.         String filePath = "inputfile.bin";
8.
9.         try (FileInputStream fis = new FileInputStream(filePath)) {
10.            int byteData;
11.            while ((byteData = fis.read()) != -1) {
12.                System.out.print((char) byteData); // Print the byte data as
```

```
            }
        } catch (FileNotFoundException e) {
            System.out.println("File not found: " + filePath);
            e.printStackTrace();
        } catch (IOException e) {
                System.err.println("An error occurred during File I/O operations.");
        }
}
```

In this program, a **FileInputStream** object is created to read from the specified file. Reading form file is achieved through **read()** method. It reads one byte at a time and returns it as an integer. When the end of the file is reached, **read()** returns **-1**.The bytes are cast to characters and printed to the console. If a specified file path is not found, it will throw **FileNotFoundException**, which is caught and handled. Any **IOException** is caught and handled. The try-with-resources statement ensures that the **FileInputStream** is closed automatically.

Writing to file using byte stream

To write data to a file using byte streams, **FileOutputStream** is the class, which is a subclass of **OutputStream**. To open a file, you just need to create an object of this class, specifying the name of the file as an argument to the constructor. This class provides many constructors, as we saw in the previous section. We can use any of them. For example, we will be using:

public FileOutputStream(String name) throws FileNotFoundException.

This constructor is used to create an output stream linked to the file specified with the given filename. The constructor takes the **String** representing the filename and will create a new file or open an existing file for writing. If the file cannot be created (e.g., due to a directory not existing, permission issues, or an invalid filename), a **FileNotFoundException** is thrown. If the file exists, it will be overwritten. The previous content will be lost.

Here is a detailed example:

```
import java.io.FileOutputStream;
import java.io.FileNotFoundException;
```

```
3. import java.io.IOException;
4.
5. public class FileOutputStreamExample {
6.     public static void main(String[] args) {
7.         String filePath = "outputfile.bin";
8.         String data = "Hello, World!";
9.
10.        try (FileOutputStream fos = new FileOutputStream(filePath)) {
11.            fos.write(data.getBytes()); // Convert the string to bytes and wr
12.        } catch (FileNotFoundException e) {
13.            System.err.println("File not found: " + filePath);
14.            } catch (IOException e) {
15.            System.err.println("An Error occurred in file I/O.");
16.        }
17.    }
18. }
```

A **FileOutputStream** object is created to write to the specified file. If the file cannot be created or opened, a **FileNotFoundException** is caught, and a message is printed. The **write(byte[] b)** method writes the bytes from the given byte array to the file. Any **IOException** is caught and handled. The try-with-resources statement ensures that the **FileOutputStream** is closed automatically.

Copying a file using byte streams

A practical example of using byte streams is copying a file. This involves reading bytes from the source file and writing them to the destination file:

```
1. import java.io.FileInputStream;
2. import java.io.FileOutputStream;
3. import java.io.IOException;
4.
5. public class FileCopyExample {
6.     public static void main(String[] args) {
7.         String inputFilePath = "inputfile.bin";
8.         String outputFilePath = "outputfile.bin";
9.
10.
            try (FileInputStream fis = new FileInputStream(inputFilePath);
```

```
11.
            FileOutputStream fos = new FileOutputStream(outputFilePath
12.
13.         byte[] buffer = new byte[1024];
14.         int bytesRead;
15.
16.         while ((bytesRead = fis.read(buffer)) != -1) {
17.             fos.write(buffer, 0, bytesRead);
18.         }
19.
20.     } catch (IOException e) {
21.         System.err.println("An Error occurred in file I/O.");
22.
23.     }
24.   }
25. }
```

In the above program, **FileInputStream** reads from the source file, and **FileOutputStream** writes to the destination file. A byte array buffer is used to read and write chunks of data, improving efficiency. The **read(byte[] b)** method reads up to **b.length** bytes into the buffer, and the **write(byte[] b, int off, int len)** method writes **len** bytes from the **buffer** starting at offset **off**. The loop continues until the end of the file is reached (**read()** returns **-1**). The try-with-resources statement ensures that both streams are closed automatically.

File input/output using character streams

Character streams in Java are designed for handling character data, which makes them suitable for text files. They operate on characters and are capable of handling different character encodings. Key classes are **FileReader**, which is used to read character data from a file and **FileWriter** for writing character data to a file. To open a file for reading or writing, we just need to create object of these classes.

Reading characters from a file using FileReader

To read data from a file using character streams, we can use **FileReaderclass**, which is a subclass of **Reader**. This class provides many constructors. We will be using the following constructor for

our program:

FileReader(String fileName)

The file which you want to open for reading, the filename, should be passed as an argument to the constructor.

Let us see an example:

```
1. import java.io.FileReader;
2. import java.io.IOException;
3.
4. public class FileReaderExample {
5.     public static void main(String[] args) {
6.         String filePath = "example.txt";
7.
8.         try (FileReader fileReader = new FileReader(filePath)) {
9.             int charData;
10.            while ((charData = fileReader.read()) != -1) {
11.                System.out.print((char) charData); // Print the character data
12.            }
13.        } catch (IOException e) {
14.            System.err.println("An Error occurred in file I/O.");
15.        }
16.    }
17. }
```

In the above program, a **FileReader** object is created to read from the specified file. The **read()** method reads one character at a time and returns it as an integer. When the end of the file is reached, **read()** returns **-1**. The characters are printed to the console by typecasting the integer value to **char**.

Writing characters to a file

To write data to a file using character streams, we can use **FileWriter** class, which is a subclass of **Writer**. This class provides many constructors. The simplest one is:

FileWriter(String fileName)

This opens the specified filename to write. If the file exists, it will be overwritten. If the file does not exist, a new file will be created.

Here is a detailed example:

```
1. import java.io.FileWriter;
```

```
import java.io.IOException;

public class FileWriterExample {
    public static void main(String[] args) {
        String filePath = "output.txt";
        String data = "Hello, World!";

        try (FileWriter fileWriter = new FileWriter(filePath)) {
            fileWriter.write(data); // Write the string data
        } catch (IOException e) {
            System.err.println("An Error occurred in file I/O.");

        }
    }
}
```

A **FileWriter** object is created to write to the specified file. The **write(String str)** method writes the string to the file. Any **IOException** is caught and handled. The try-with-resources statement ensures that the **FileWriter** is closed automatically.

Reading using BufferedReader

As we performed console input by using the **BufferedReader** class, we can read from a file in the same manner. **FileReader** is a class in the **java.io** package. It is a convenient class for reading character files. The **FileReader** object reads characters from a file specified by the **filePath**. **BufferedReader** is a class in the **java.io** package. It wraps another **Reader** (like **FileReader**) to buffer the input, improving efficiency by reducing the number of I/O operations. It provides methods to read text, arrays, and lines. The **FileReader** class provides the following basic constructor:

FileReader(filePath):

This creates a new instance of **FileReader** that reads from the file specified by **filePath**. The **FileReader** constructor takes a file path as a string argument, which is the location of the file to be read.

The **BufferedReader** class provides a constructor as follows:

BufferedReader(new FileReader(filePath)):

This creates a new instance of **BufferedReader** that wraps the

FileReader. Wrapping **FileReader** with **BufferedReader** enhances efficiency by adding buffering capabilities and provides convenient methods like **readLine()**. It is convenient to create a **FileReader** to read the contents of a file and wrap it in a **BufferedReader** to buffer the input and read lines efficiently.

Here is a complete example of reading from a file using **BufferedReader**:

Example: Reading from a **File** using **BufferedReader**

```
1. import java.io.BufferedReader;
2. import java.io.FileReader;
3. import java.io.IOException;
4.
5. public class FileReadingExample {
6.     public static void main(String[] args) {
7.         String filePath = "example.txt";
8.
9.         try (BufferedReader reader = new BufferedReader(new FileReade
10.            String line;
11.            while ((line = reader.readLine()) != null) {
12.                System.out.println(line);
13.            }
14.        } catch (IOException e) {
15.            System.err.println("Error reading the file: " + e.getMessage());
16.        }
17.    }
18. }
```

The **filePath** variable holds the path to the file to be read. A **BufferedReader** is wrapped around a **FileReader** to read text from the file efficiently, line by line. Loop ensures that the **readLine()** method reads each line from the file until the end of the file is reached. An **IOException** is caught and handled to deal with potential issues like the file not being found.

Writing lines to a file using BufferedWriter

BufferedWriter provides buffering for efficient writing of characters, arrays, and lines. A **BufferedWriter** object can be created, wrapping a **FileWriter** for the specified file using following constructors.

FileWriter(String fileName)

This **FileWriter** constructor takes the file name as an argument and opens it for output.

BufferedWriter(Writer out)

This constructor for **BufferedWriter** takes a **Writer** object as an argument, typically a **FileWriter**.

Both can be combined together to get a **BufferedWriter** object wrapped on a file to write.

Here is a detailed example:

```
1. import java.io.BufferedWriter;
2. import java.io.FileWriter;
3. import java.io.IOException;
4.
5. public class BufferedWriterExample {
6.     public static void main(String[] args) {
7.         String filePath = "output.txt";
8.
9.         try (BufferedWriter bufferedWriter = new BufferedWriter(new Fil
10.            bufferedWriter.write("Hello, World!");
11.            bufferedWriter.newLine(); // Add a new line
12.            bufferedWriter.write("Welcome to file handling in Java.");
13.        } catch (IOException e) {
14.            System.err.println("An Error occurred in file I/O.");
15.
16.        }
17.    }
18. }
```

The **filePath** variable holds the path to the file to be written. A **BufferedWriter** is wrapped around a **FileWriter** to write text to the file efficiently. The **write()** method writes the content to the file. The **newLine()** method writes a newline character. An **IOException** is caught and handled to deal with potential issues like the inability to write to the file.

Using PrintWriter for simplified writing

PrintWriter provides convenient methods to write formatted text to a file. The **PrintWriter** class provides a constructor.

PrintWriter(String fileName)

This creates a **PrintWriter** object to write to a file with the specified name. If the file exists, it will be overwritten. If the file does not exist, a new file will be created. Let us see one example of writing to file using **PrintWriter**.

Example: Using **PrintWriter**:

```
1. import java.io.PrintWriter;
2. import java.io.IOException;
3.
4. public class PrintWriterExample {
5.     public static void main(String[] args) {
6.         String filePath = "output.txt";
7.
8.         try (PrintWriter writer = new PrintWriter(filePath)) {
9.             writer.println("Hello, world!");
10.            writer.printf("Formatted number: %.2f%n", 123.456);
11.        } catch (IOException e) {
12.            System.err.println("Error writing to the file: " + e.getMessage()
13.        }
14.    }
15. }
```

A **PrintWriter** is used to write text to the file, with methods like **println()** and **printf()** for formatted output. The **println()** method writes a line of text, and the **printf()** method writes formatted text.

Automatic file closing

Automatic file closing in Java is achieved using the try-with-resources statement introduced in Java 7. This feature ensures that each resource declared in the try-with-resources statement is closed automatically at the end of the statement, making code more readable and less error-prone by eliminating the need to explicitly close resources.

The try-with-resources statement is a try statement that declares one or more resources. A resource is an object that must be closed after the program is finished, such as files or streams. The resource must implement the **AutoCloseable** interface, which includes the **close()** method. When the try block finishes, whether normally or

due to an exception, the resources are closed automatically.

Syntax of try-with-resources:

```
try (ResourceType resource = new ResourceType()) {
    // Use the resource
} catch (ExceptionType e) {
    // Handle exceptions
}
```

In this statement, **ResourceType** refers to the type of the resource, which must implement the **AutoCloseable** interface. The resource is the variable that declares and initializes this resource.

File reading using try-with-resources

Let us look at a detailed example of reading a file using **BufferedReader** with automatic resource management:

```
1. import java.io.BufferedReader;
2. import java.io.FileReader;
3. import java.io.IOException;
4.
5. public class FileReadingExample {
6.     public static void main(String[] args) {
7.         String filePath = "example.txt";
8.
9.         // Try-with-
   resources statement to ensure automatic resource management
10.        try (BufferedReader bufferedReader = new BufferedReader(new F
11.            String line;
12.            while ((line = bufferedReader.readLine()) != null) {
13.                System.out.println(line);
14.            }
15.
16.        } catch (IOException e) {
17.            System.err.println("Error while reading from
   the file: " + e.getMessage());
18.        }
19.    }
20. }
```

In the above example, Resource declaration is done with,

try (BufferedReader bufferedReader = new BufferedReader(new FileReader(filePath)))

Here, **BufferedReader** is created with a **FileReader** to read from the file specified by **filePath**. Both **BufferedReader** and **FileReader** implement the **AutoCloseable** interface, ensuring they will be closed automatically.

Using the resource can be demonstrated with the statements:

```
String line;
while ((line = bufferedReader.readLine()) != null) {
   System.out.println(line);
}
```

This block reads lines from the file and prints them to the console. If an exception occurs during this process, the catch block handles it.

After the try block finishes (whether it exits normally or due to an exception), the **BufferedReader** and **FileReader** are automatically closed, ensuring no resource leaks occur.

Multiple resources in try-with-resources

You can declare multiple resources in a single try-with-resources statement by separating them with a semicolon (Please refer to the example program for *copying a file using byte streams* above). Each resource will be closed automatically at the end of the statement.

Resource declaration can be done as follows:

```
try (FileInputStream fis = new FileInputStream(sourcePath);
    FileOutputStream fos = new FileOutputStream(destPath))
```

Here, both **FileInputStream** and **FileOutputStream** are declared within the try-with-resources statement. After the try block finishes, both **FileInputStream** and **FileOutputStream** are automatically closed, ensuring no resource leaks occur.

Benefits of try-with-resources

The following are the benefits of try-with-resources:

- **Automatic resource management**: Ensures resources are closed automatically, reducing the risk of resource leaks.
- **Simplified code**: Eliminates the need for explicit finally blocks to close resources, making code cleaner and easier to read.

- **Improved exception handling**: Allows for more efficient and consistent exception handling.

Conclusion

In this chapter, we covered how to perform input-output operations in Java. We discussed about the role of streams, and their various types, including byte and character streams. We discussed various classes of java.io package in detail with their use cases. We explored standard I/O operations that utilize the system's default input and output streams, and delved into file I/O, demonstrating how to read from and write to files effectively. This chapter also discussed an example of reading through Scanner class, which is a part of java.util package. Additionally, we highlighted the importance of automatic resource management using the try-with-resources statement, which ensures that resources are properly closed, thus preventing potential resource leaks.

In the next chapter, we are going to learn about concurrent programming in Java.

Exercises

1. Write a Java program that prompts the user to enter their name and age, then prints out a greeting message with their name and age.
2. Write a program that reads a text file (input.txt) and prints its contents to the console.
3. Create a program that asks the user to enter several lines of text until they type "exit". Save these lines to a file (output.txt).
4. Write a program that uses Scanner to read integers from the user until they enter a negative number. Calculate and print the average of the positive numbers entered.
5. Modify one of the above programs to handle FileNotFoundException and IOException gracefully using try-catch blocks.
6. Implement a program that reads a large text file (bigfile.txt)

using BufferedReader and counts the number of lines in it.

7. Write a program that copies the contents of one text file (source.txt) to another (destination.txt).

Chapter 8

Concurrent Programming

Introduction

Multithreading is a powerful feature in Java that allows concurrent execution of two or more parts of a program to maximize the utilization of CPU. Each part of such a program is called a thread, and the process of executing multiple threads simultaneously is known as multithreading. Multithreading is particularly useful in applications where tasks can be performed independently, such as web servers, gaming applications, and real-time systems.

In this chapter, we will explore the fundamentals of multithreading in Java, starting with an understanding of the basic concepts and progressing through the creation and management of threads. This chapter transitions to concurrent programming and explores multithreading in Java, which allows programs to execute multiple tasks concurrently. Readers will learn how to create and manage threads, synchronize access to shared resources, and coordinate thread execution, enhancing the responsiveness and efficiency of Java applications.

Structure

The chapter covers the following topics:

- Multithreading fundamentals
- Thread class
- Runnable interface
- Understanding the main Thread
- Creating a Thread
- The life cycle of Thread
- Implementation of thread methods
- Synchronization
- Communication between threads

Objectives

By the end of this chapter, you will be able to understand the fundamentals of multithreading, including its advantages and the distinction between processes and threads. You will learn to create threads using the Thread class and the Runnable interface, recognizing the differences between these approaches. You will gain a comprehensive understanding of the thread life cycle and the transitions between its various states. You will be able to develop applications that manage single and multiple threads, set thread priorities, and organize threads into groups. Additionally, you will become proficient in using key thread methods such as start(), run(), sleep(), join(), and interrupt(), and apply best practices for their implementation. Finally, you will grasp the importance of synchronization in multithreaded programs, employing synchronized methods and statements to control access to critical sections, while implementing strategies to avoid deadlocks and ensure thread safety. This knowledge will enable you to develop efficient, reliable, and scalable multithreaded applications in Java.

Multithreading fundamentals

The programs which we learned so far were working in a single flow of execution. Another approach is multiple tasks running

concurrently. Concurrent programming is a computing paradigm where multiple computations are executed during overlapping time periods rather than sequentially. This approach leverages the capabilities of modern multi-core processors to enhance performance and responsiveness in software applications. In concurrent programming, tasks can be structured as threads, processes, or asynchronous operations that run independently yet may interact or share resources. Proper management of these interactions is critical to avoid issues such as race conditions, deadlocks, and resource contention. Synchronization techniques, like mutexes, semaphores, and condition variables, are employed to ensure safe and predictable execution of concurrent tasks. Concurrent programming is essential in developing responsive user interfaces, real-time systems, high-performance computing applications, and network servers, allowing them to handle multiple tasks simultaneously and efficiently.

Multithreading is a programming technique where multiple threads are spawned by a process to execute tasks concurrently, improving performance and resource utilization. Each thread runs independently while sharing the process's resources, such as memory and file handles. This allows for parallel execution, enabling more efficient use of multi-core processors. Key concepts in multithreading include thread creation, synchronization, and communication. Synchronization mechanisms like locks, semaphores, and monitors are crucial to prevent race conditions and ensure data consistency when multiple threads access shared resources. Multithreading is widely used in applications requiring concurrent operations, such as web servers, real-time simulations, and complex computations. The multithreading concept in java is implemented by thread class and runnable interface.

Thread class

Java provides a class named **Thread**, which is part of the **java.lang** package. It is used to create and manage threads. It represents a thread of execution. Each thread represents a separate path of execution in a program. The main purpose of the **Thread** class is to allow Java programs to perform multiple tasks simultaneously, known as concurrency. By creating multiple threads, a program can

execute different parts of code concurrently, enabling tasks to run independently of each other.

Thread class provides many methods for managing threads and controlling their behavior in a multithreaded Java application. Commonly used methods are listed below:

Method	Description
start()	Starts the thread, invoking the **run()** method in a separate call stack.
run()	Contains the code that constitutes the new thread's task; typically overridden in a subclass.
sleep(long millis)	Causes the currently executing thread to sleep for the specified number of milliseconds.
sleep(long millis, int nanos)	Causes the currently executing thread to sleep for the specified number of milliseconds plus nanoseconds.
join()	Waits for the thread to die.
join(long millis)	Waits for the specified time for the thread to die.
join(long millis, int nanos)	Waits for the specified time plus nanoseconds for the thread to die.
interrupt()	Interrupts the thread.
isInterrupted()	Tests whether the thread has been interrupted.
interrupted()	Tests whether the current thread has been interrupted (static method).
getId()	Returns the thread's ID.
getName()	Returns the thread's name.
setName(String name)	Changes the thread's name.
getPriority()	Returns the thread's priority.
setPriority(int priority)	Changes the thread's priority.
getState()	Returns the state of the thread (e.g., **NEW**, **RUNNABLE**, **BLOCKED**, **WAITING**, **TIMED_WAITING**, **TERMINATED**).
isAlive()	Tests if the thread is alive.
yield()	Causes the currently executing thread object to temporarily pause and allow other threads to execute.
currentThread()	Returns a reference to the currently executing thread object (static method).
setDaemon(boolean on)	Marks this thread as either a daemon thread or a user thread.
isDaemon()	Tests if this thread is a daemon thread.

***Table 8.1**: Commonly used methods of Thread class*

Runnable interface

The **Runnable** interface in Java is a functional interface that represents a task that can be executed by a thread. It is part of the **java.lang** package and contains a single abstract method, **run()**, which must be implemented by any class that implements the interface. The **Runnable** interface is often used when you want to create a thread, but the class you are working with should not or

cannot extend the **Thread** class (since Java does not support multiple inheritance). The **Runnable** interface has only one method:

public void run();

This method contains the code that constitutes the new thread's task.

Understanding the main Thread

When a Java program begins execution, the main thread in Java is the initial thread that starts. The **main** method, defined as **public static void main(String[] args)**, is the entry point of any Java application. The JVM calls this method to start the program, creating the main thread. Every Java application runs at least one thread, the main thread, which executes the main method. Additional threads can be created by instantiating the **Thread** class or implementing the *Runnable* interface. The main thread runs with a default priority (usually 5). The main thread's life cycle ends when the main method completes its execution. The main thread can spawn new threads. You can obtain a reference to the main thread using **Thread.currentThread()**. You can control the main thread's execution using methods like **sleep()**, **yield()**, and **join()**.

Here is a simple example illustrating the main thread:

```
1. public class MainThreadExample {
2.     public static void main(String[] args) {
3.         // Displaying the current thread
4.         Thread mainThread = Thread.currentThread();
5.         System.out.println("Current Thread: " +
mainThread.getName());
6.
7.
8.         // Changing the name of the main thread
9.         mainThread.setName("Primary Thread");
10.        System.out.println("New Thread Name: " +
mainThread.getName());
11.
12.
13.        // Displaying the priority of the main thread
14.        System.out.println("Main Thread Priority: " +
```

```
        mainThread.getPriority());
15.
16.
17.         // Making the main thread sleep for 1 second
18.         try {
19.             Thread.sleep(1000);
20.         } catch (InterruptedException e) {
21.             System.out.println("Main thread interrupted");
22.         }
23.
24.
25.         System.out.println("Main thread finished execution");
26.     }
27. }
```

The program will give the following output:

Current Thread: main
New Thread Name: Primary Thread
Main Thread Priority: 5
Main thread finished execution

You can control the main thread like any other **Thread** in Java but be cautious as it directly affects the program's execution. While you can set priorities for threads, it is generally recommended to let the JVM handle thread scheduling. The main thread's lifecycle is tied to the completion of the **main** method. When the **main** method exits, the **main** thread terminates unless other user threads are running.

Creating a Thread

A thread can be created by instantiating the **Thread** class. You can define your own thread by using two alternate methods, either by extending **Thread** class or by implementing *Runnable* interface. Let us explore each method with its use case.

Extending the Thread class

To create a new thread, you need to extend the **Thread** class and override its **run()** method. This method contains the code that defines the task to be executed by the thread.

Here is an example of creating a thread by extending the **Thread**

class:

```
class TestThread extends Thread {
    public void run() {
        System.out.println("Thread is running.");
    }
}
```

Now, an instance of the **TestThread** class can be created and start the thread:

```
public class SingleThreadExample {
public static void main(String[] args) {
TestThread t1 = new TestThread();
t1.start();
}
}
```

TestThread extends the **Thread** class and overrides its run method. In the main method of **SingleThreadExample**, a new instance of **TestThread** is created with the name "**t1**". The start method is called on this instance, which in turn calls the run method in a new thread of execution. The run method prints **Thread is running**.

By using Runnable interface

To create a thread using the **Runnable** interface in Java, create a class that implements the **Runnable** interface and override its run method. Instantiate a **Thread** object, passing the **Runnable** implementation to its constructor. Call the start method on the **Thread** object to begin execution.

Here is an example:

```
class TestRunnable implements Runnable {
    public void run() {
        System.out.println("Thread is running.");
    }
}

```

Create an instance of **Thread** with a **MyRunnable** object and start the thread:

```
public class SingleThreadExample {
    public static void main(String[] args) {

```

```
        TestRunnable myRunnable = new TestRunnable();
        Thread t1 = new Thread(myRunnable);
        t1.start();
    }
}
```

The **TestRunnable** class implements the **Runnable** interface and overrides the run method to define the code to be executed by the thread. In the main method of **SingleThreadExample**, a new **TestRunnable** instance is created with the name "**myRunnable**". A new **Thread** object is created, passing the **myRunnable** instance to its constructor. The start method is called on the **Thread** object, which begins execution of the run method in a new thread.

The program will print the message as the previous implementation:

Thread is running

Implementing the **Runnable** interface is generally considered the preferred approach for several reasons. Implementing **Runnable** separates, the task to be performed from the thread management. This makes the code more modular and easier to maintain. The **Runnable** implementation focuses solely on the task, while the **Thread** class handles the threading mechanism. It also solves multiple inheritance limitations of java. Java does not support multiple inheritance, meaning a class can only extend one other class. By implementing **Runnable**, you can still create a thread while allowing your class to extend another class if needed.

Creating multiple Thread

In our previous program, we saw only two threads, one main thread, and one child thread. However, there can be many threads in a program as it is required. Creating multiple threads involves creating multiple instances of the **Thread** class or **Runnable** interface and starting them.

Let us see an example of creating multiple threads:

```
class MyThread extends Thread {
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(Thread.currentThread().getId() + " Value: "
            try {
```

```
            Thread.sleep(500); // Sleep for 500 milliseconds
        } catch (InterruptedException e) {
            System.out.println(e);
        }
    }
}

public class ThreadExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();

        t1.start(); // Start the first thread
        t2.start(); // Start the second thread
    }
}
```

In this example **MyThread** class extends **Thread** and overrides the run method, which contains the code to be executed by the thread. **t1.start()** and **t2.start()** initiate the execution of the threads, running the run method concurrently. When you run the **ThreadExample** class, which starts two instances of **MyThread** (t1 and t2), each thread will execute its **run()** method independently. The **run()** method contains a loop that prints the thread ID and a counter value from 0 to 4, with a delay of 500 milliseconds between each iteration. Here is how the output might look:

Thread-0 Value: 0
Thread-1 Value: 0
Thread-0 Value: 1
Thread-1 Value: 1
Thread-0 Value: 2
Thread-1 Value: 2
Thread-0 Value: 3
Thread-1 Value: 3
Thread-0 Value: 4
Thread-1 Value: 4

In the above execution, **Thread-0** and **Thread-1** are the default names given to the threads by Java (since **MyThread** extends

Thread and does not override the **Thread** class's **getName()** method). Each thread executes its **run()** method independently, hence you see interleaved output from both threads. The **Thread.currentThread().getId()** method returns a unique ID for each thread. The exact interleaving of output between **Thread-0** and **Thread-1** can vary because it depends on the operating system's thread scheduling mechanism. However, each thread will print its sequence of values from 0 to 4, with a delay of 500 milliseconds between each print due to the **Thread.sleep(500)** call inside the loop.

Let us implement the same task using the **Runnable** interface. Here is an example demonstrating the creation of multiple threads using the **Runnable** interface:

```
1. class MyRunnable implements Runnable {
2.     public void run() {
3.         for (int i = 0; i < 5; i++) {
4.             System.out.println(Thread.currentThread().getName() + " Valu
5.             try {
6.                 Thread.sleep(500); // Sleep for 500 milliseconds
7.             } catch (InterruptedException e) {
8.                 System.out.println(e);
9.             }
10.        }
11.    }
12. }
13.
14. public class RunnableExample {
15.     public static void main(String[] args) {
16.         MyRunnable myRunnable = new MyRunnable();
17.
18.         Thread t1 = new Thread(myRunnable);
19.         Thread t2 = new Thread(myRunnable);
20.
21.         t1.start(); // Start the first thread
22.         t2.start(); // Start the second thread
23.     }
24. }
```

The **MyRunnable** class implements the **Runnable** interface and

overrides the run method. This method contains the code to be executed by the thread. In the **RunnableExample** class, instances of **Thread** are created and passed an instance of **MyRunnable**. The **start** method is called on the Thread objects (**t1** and **t2**), which causes the **run** method of **MyRunnable** to be executed concurrently in separate threads. By using the Runnable interface, you can create a class that is designed to be executed by a thread without having to extend the Thread class.

The life cycle of thread

The life cycle of a thread in Java includes several states, which represent the various stages of a thread's execution. These states are:

- **New**: The thread has been created but has not yet started. This is the initial state after a Thread object is instantiated.
- **Runnable**: The thread is ready to run and waiting for the CPU to be available. In this state, the thread may be running or it might be waiting for CPU time.
- **Blocked**: The thread is blocked and waiting for a monitor lock to enter a synchronized block/method or re-enter a synchronized block/method after calling **Object.wait()**.
- **Waiting**: The thread is waiting indefinitely for another thread to perform a particular action. For example, the thread is waiting due to calling **Object.wait()** without a timeout, or **Thread.join()** without a timeout.
- **Timed waiting**: The thread is waiting for another thread to perform a specific action within a specified waiting time. For example, the thread is waiting due to calling **Thread.sleep()**, **Object.wait()** with a timeout, or **Thread.join()** with a timeout.
- **Terminated**: The thread has completed its execution and has exited. This is the final state of a thread.

Life cycle methods

To control the life cycle of a thread, Java provides several methods:

- **start()**: Starts the execution of the thread by invoking its run

method.

- **run()**: Contains the code that constitutes the new thread's task. This method should be overridden by any class that implements **Runnable**.
- **sleep(long millis)**: Puts the thread into a timed waiting state for the specified duration.
- **join()**: Waits for the thread to die. The calling thread waits until the thread on which join was called terminates.
- **yield()**: Causes the currently executing thread to temporarily pause and allow other threads to execute.
- **wait()**: Causes the current thread to wait until another thread invokes the **notify()** or **notifyAll()** methods for the same object.
- **notify()**: Wakes up a single thread that is waiting on the object's monitor.
- **notifyAll()**: Wakes up all threads that are waiting on the object's monitor.

Here is a diagram to visualize the states and transitions of a thread's life cycle:

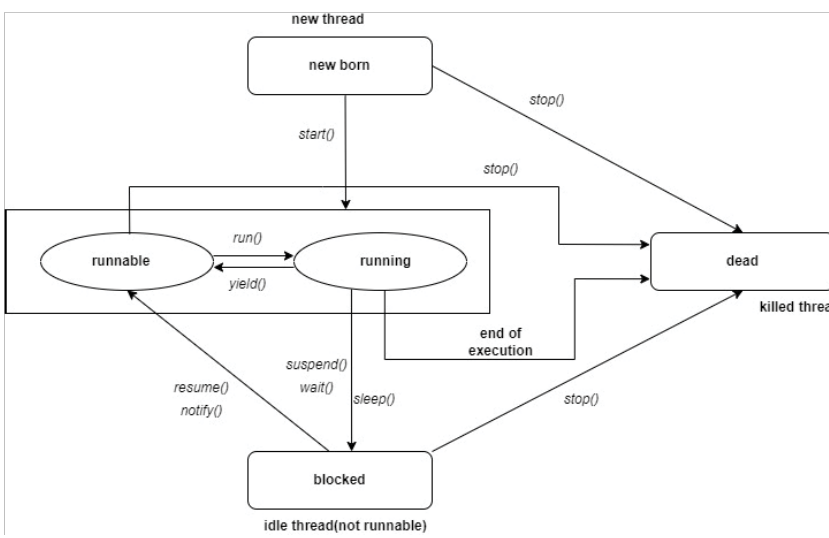


***Figure 8.1**: Lifecycle of thread*

Implementation of thread methods

Java's Thread class provides numerous methods to control and manage the behavior of threads. The following section describes the purpose, implementation, and usage of these methods:

- **start()**

  Purpose: Initiates the thread's execution, causing the JVM to call the thread's **run()** method.

  Usage:

  ```
  Thread t1 = new Thread(new MyRunnable());
  t1.start();
  ```

- **run()**

  Purpose: Contains the code that constitutes the new thread's task. It should be overridden in a subclass of **Thread** or in a class implementing **Runnable**.

  Usage:

  ```
  class MyRunnable implements Runnable {
      public void run() {
          System.out.println("Thread is running.");
      }
  }
  ```

- **sleep(long millis)**

  Purpose: Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

  Usage:

  ```
  try {
      Thread.sleep(1000); // Sleep for 1000 milliseconds (1 second)
  } catch (InterruptedException e) {
      e.printStackTrace();
  }
  ```

- **join()**

  Purpose: Waits for the thread on which it is called to die. There are overloaded versions that allow specifying a

maximum time to wait.

Usage:

```
Thread t1 = new Thread(new MyRunnable());
t1.start();
try {
   t1.join(); // Main thread will wait until t1 finishes
} catch (InterruptedException e) {
   e.printStackTrace();
}
```

- **yield()**

  Purpose: Causes the currently executing thread to temporarily pause and allow other threads to execute.

  Usage:

  ```
  Thread.yield(); // Suggests that the current thread is willing to yield its current use of a processor
  ```

- **getName()**

  Purpose: Returns the name of the thread.

  Usage:

  ```
  Thread t1 = new Thread(new MyRunnable());
  System.out.println(t1.getName()); // Default name like Thread-0
  ```

- **setName(String name)**

  Purpose: Sets the name of the thread.

  Usage:

  ```
  Thread t1 = new Thread(new MyRunnable());
  t1.setName("MyThread");
  ```

- **getPriority()**

  Purpose: Returns the priority of the thread.

  Usage:

  ```
  Thread t1 = new Thread(new MyRunnable());
  int priority = t1.getPriority(); // Default priority is 5 (NORM_PRIORITY)
  ```

- **setPriority(int priority)**

  Purpose: Sets the priority of the thread. The thread priority

is represented by an integer from 1 to 10.

Usage:

Thread t1 = new Thread(new MyRunnable());
t1.setPriority(Thread.MAX_PRIORITY); // Sets priority to 10

- **isAlive()**

  Purpose: Tests if the thread is alive (a thread is alive if it has been started and has not yet died).

  Usage:

  Thread t1 = new Thread(new MyRunnable());
  t1.start();
  boolean isAlive = t1.isAlive();

- **isDaemon()**

  Purpose: Tests if the thread is a daemon thread.

  Usage:

  Thread t1 = new Thread(new MyRunnable());
  boolean isDaemon = t1.isDaemon();

- **setDaemon(boolean on)**

  Purpose: Marks the thread as a daemon thread or a user thread. A daemon thread is a background thread that does not prevent the JVM from exiting when the program finishes.

  Usage:

  Thread t1 = new Thread(new MyRunnable());
  t1.setDaemon(true);

Let us see one example using various **Thread** methods:

```
1. class TestRunnable implements Runnable {
2.    public void run() {
3.       for (int i = 0; i < 5; i++) {
4.          try {
5.             Thread.sleep(500); // Sleep for 500 milliseconds
6.          } catch (InterruptedException e) {
7.             System.out.println(e);
8.          }
9.          System.out.println(Thread.currentThread().getName()
```

```
        + " is running, iteration: " + i);
10.         }
11.     }
12. }
13.
14. public class ThreadMethodsExample {
15.     public static void main(String[] args) {
16.         Thread t1 = new Thread(new TestRunnable(), "Thread
    1");
17.         Thread t2 = new Thread(new TestRunnable(), "Thread
    2");
18.
19.         t1.start();
20.         t2.start();
21.
22.         try {
23.             t1.join(); // Wait for t1 to finish
24.         } catch (InterruptedException e) {
25.             System.out.println(e);
26.         }
27.
28.         System.out.println(t1.getName() + " is alive: " +
    t1.isAlive());
29.         System.out.println(t2.getName() + " is alive: " +
    t2.isAlive());
30.
31.         t2.setPriority(Thread.MAX_PRIORITY); // Set the highest
    priority
32.         System.out.println(t2.getName() + " priority: " +
    t2.getPriority());
33.
34.         t2.setDaemon(true); // Set t2 as a daemon thread
35.         System.out.println(t2.getName() + " is daemon: " +
    t2.isDaemon());
36.
37.         Thread.yield(); // Yield execution to other threads
38.     }
39. }
40.
```

In the above example program, we create two threads (**t1** and **t2**)

with names "**Thread 1**" and "**Thread 2**", respectively, and start them using **start()**. Each thread sleeps for 500 milliseconds in each iteration of its loop within the **run()** method. The main thread waits for t1 to finish using **join()** before proceeding. We check if both threads are alive using **isAlive()** after t1 has finished. We set the priority of **t2** to the maximum value (10) using **setPriority()** and then retrieve it using **getPriority()**. We set t2 as a daemon thread using **setDaemon(true)** and check its daemon status using **isDaemon()**. We call **Thread.yield()** to suggest the current thread is willing to yield its execution.

The output of the provided Java program will depend on the timing of thread execution and the behavior of thread scheduling, but we can predict the possible output based on the structure of the program and typical thread behavior:

Thread 1 is running, iteration: 0
Thread 2 is running, iteration: 0
Thread 1 is running, iteration: 1
Thread 2 is running, iteration: 1
Thread 1 is running, iteration: 2
Thread 2 is running, iteration: 2
Thread 1 is running, iteration: 3
Thread 2 is running, iteration: 3
Thread 1 is running, iteration: 4
Thread 2 is running, iteration: 4
Thread 1 is alive: false
Thread 2 is alive: false
Thread 2 priority: 10
Thread 2 is daemon: true

Synchronization

Synchronization in Java is used to control the access of multiple threads to shared resources. When multiple threads access a shared resource (like a variable or an object), there is a risk of inconsistent or incorrect results due to concurrent access. Synchronization is a way to control this access to ensure that only one thread can access a resource at a time. This helps avoid issues such as race conditions, where the outcome depends on the unpredictable order of thread

execution. Without proper synchronization, it is possible for one thread to modify a shared variable while another thread is in the middle of using or updating that same variable, which can lead to inconsistent or incorrect results. Java provides synchronized keywords to achieve synchronization. Here are some scenarios where synchronization is essential:

- **Race conditions**: Multiple threads accessing a resource without proper synchronization may lead to inconsistent data.
- **Visibility issues**: Changes made by one thread may not be visible to other threads.
- **Atomicity problems**: Operations that should be atomic (indivisible) may not behave correctly when interrupted.

Java provides two primary mechanisms for synchronization: Synchronized methods, synchronized statements (or blocks).

Synchronized methods

A method can be synchronized by mentioning the synchronized keyword in its declaration. A synchronized method ensures that only one thread can execute it at a time for a given instance. If a thread is executing a synchronized method, all other threads that invoke synchronized methods on the same object are blocked until the first thread completes. The general syntax of declaring a synchronized method is:

synchronized returnType methodName(argumentList)

Let us see one example program:

```
1. class Table {
2.     synchronized void printTable(int n) { // synchronized method
3.         for (int i = 1; i <= 5; i++) {
4.             System.out.println(n * i);
5.             try {
6.                 Thread.sleep(400);
7.             } catch (InterruptedException e) {
8.                 System.out.println(e);
9.             }
10.        }
11.    }
```

```
}

class MyThread1 extends Thread {
    Table t;

    MyThread1(Table t) {
        this.t = t;
    }

    public void run() {
        t.printTable(5);
    }
}

class MyThread2 extends Thread {
    Table t;

    MyThread2(Table t) {
        this.t = t;
    }

    public void run() {
        t.printTable(10);
    }
}

public class TestSynchronization {
    public static void main(String[] args) {
        Table obj = new Table(); // only one object
        MyThread1 t1 = new MyThread1(obj);
        MyThread2 t2 = new MyThread2(obj);
        t1.start();
        t2.start();
    }
}
```

In this example, the **printTable** method is synchronized. This means if one thread is executing it, the other thread cannot enter until the first one finishes. This ensures that the multiplication table is printed correctly without interleaving lines from both threads. The

provided Java program involves two threads (**MyThread1** and **MyThread2**) accessing a synchronized method (**printTable**) of the **Table** class. Here is the expected output based on synchronized method behavior and thread execution:

5
10
15
20
25
10
20
30
40
50

MyThread1 and **MyThread2** are both started and they both call the **printTable** method on the same instance of Table (**obj**). Since **printTable** is synchronized, when **t1** (which runs **MyThread1**) starts executing **t.printTable(5)**, **t2** (which runs **MyThread2**) has to wait until t1 finishes its execution of the synchronized method. **t1** (running **MyThread1**) prints the multiples of 5: 5, 10, 15, 20, 25. During this time, **t2** (running **MyThread2**) waits because the **printTable** method is synchronized. After **t1** completes its execution of **printTable**, **t2** is allowed to execute the **printTable** method. Hence, **t2** prints the multiples of 10: 10, 20, 30, 40, 50.

Synchronized statements or blocks

Synchronized statements (or blocks) provide a more fine-grained approach to synchronization compared to synchronized methods, allowing you to lock specific code blocks rather than entire methods. Synchronized statements (or blocks) are used to control access to a critical section of code by ensuring that only one thread can execute that section at a time. The synchronized keyword is used to mark a block of code that should be executed by only one thread at a time. Synchronized statements use the synchronized keyword inside a method to lock an object for a specific block of code.

The basic syntax of a synchronized block is:

```
synchronized(lockObject) {
   // Critical section code
}
```

lockObject is the object on which the thread acquires the lock. The lock ensures that only one thread can execute the synchronized block code at a time for the given **lockObject**. When a thread enters a synchronized block, it must acquire the lock associated with the **lockObject**. If another thread already holds the lock, the thread attempting to enter the block will be suspended until the lock becomes available. Let us modify the above program to use a synchronized block:

```
1. class Table {
2.     void printTable(int n) {
3.         synchronized (this) { // synchronized block
4.             for (int i = 1; i <= 5; i++) {
5.                 System.out.println(n * i);
6.                 try {
7.                     Thread.sleep(400);
8.                 } catch (InterruptedException e) {
9.                     System.out.println(e);
10.                }
11.            }
12.        }
13.    }
14. }
15.
16. class MyThread1 extends Thread {
17.    Table t;
18.
19.    MyThread1(Table t) {
20.        this.t = t;
21.    }
22.
23.    public void run() {
24.        t.printTable(5);
25.    }
26. }
27.
28. class MyThread2 extends Thread {
```

```
    Table t;

    MyThread2(Table t) {
        this.t = t;
    }

    public void run() {
        t.printTable(10);
    }
}

public class TestSynchronization {
    public static void main(String[] args) {
        Table obj = new Table(); // only one object
        MyThread1 t1 = new MyThread1(obj);
        MyThread2 t2 = new MyThread2(obj);
        t1.start();
        t2.start();
    }
}
```

In this example, only the block of code inside the synchronized statement is synchronized. This ensures that the critical section of code is executed by only one thread at a time. In this case, the shared resource is an instance of the **Table** class, and two threads are attempting to use it simultaneously. The **Table** class has a method **printTable(int n)**, which prints the multiplication table of the given number **n**.

The **printTable** method is synchronized using a synchronized block on this (the current instance of **Table**). Both **MyThread1** and **MyThread2** extend the **Thread** class. Each thread takes a **Table** object in its constructor and stores it in the **t** attribute. The **run** method of each thread calls the **printTable** method on the **Table** object with different arguments (5 and 10, respectively). In the **main** method, a single **Table** object (obj) is created. Two threads (**t1** and **t2**) are created, both using the same Table object. Both threads are started with **t1.start()** and **t2.start()**. When the program runs, **t1** and **t2** start almost simultaneously. Since **printTable** is synchronized on this, only one thread can execute the synchronized block at a time on the same **Table** instance. Let us

assume **t1** acquires the lock first. It prints the table of 5. Each line is printed with a **400 ms** delay due to **Thread.sleep(400)**. After **t1** completes, it releases the lock, and **t2** can then acquire the lock and print the table of 10.

In summary, the synchronization ensures that the multiplication tables are printed sequentially without interleaving, even though multiple threads are trying to access the same method concurrently.

Both synchronization methods help to prevent thread interference and memory consistency errors. Choosing between synchronized methods and synchronized blocks depends on the granularity of synchronization you need and the performance implications.

You can combine both synchronized methods and synchronized blocks in your program, depending on your needs. For instance, you might use a synchronized block inside a non-synchronized method to protect only a critical section of code:

```
class Counter {
    private int count = 0;

    public void increment() {
        synchronized (this) {
            count++;
        }
    }

    public synchronized int getCount() {
        return count;
    }
}

public class TestSynchronization {
    public static void main(String[] args) {
        Counter counter = new Counter();

        Thread t1 = new Thread(new Runnable() {
            public void run() {
                for (int i = 0; i < 1000; i++) {
                    counter.increment();
                }
```

```
24.             }
25.         });
26.
27.         Thread t2 = new Thread(new Runnable() {
28.             public void run() {
29.                 for (int i = 0; i < 1000; i++) {
30.                     counter.increment();
31.                 }
32.             }
33.         });
34.
35.         t1.start();
36.         t2.start();
37.
38.         try {
39.             t1.join();
40.             t2.join();
41.         } catch (InterruptedException e) {
42.             e.printStackTrace();
43.         }
44.
45.         System.out.println("Count: " + counter.getCount());
46.     }
47. }
```

The above code demonstrates a multithreading scenario where two threads increment a shared counter. The **Counter** class uses synchronization to ensure that increments are performed safely when accessed by multiple threads. The **Counter** class has a private integer count initialized to 0. The increment method increments count within a synchronized block to ensure thread safety. The **getCount** method returns the value of count and is marked as synchronized for consistent visibility across threads. A **Counter** object is created. Two threads (**t1** and **t2**) are created using anonymous **Runnable** implementations. Each thread calls the increment method 1000 times. Both threads are started. The **join** method is used to wait for both threads to complete their execution before printing the final count. The final value of count is printed. Both threads start almost simultaneously and execute the increment method 1000 times each. Due to the synchronization in the

increment method, only one thread can increment the counter at a time. The **main** method waits for both threads to finish using **join**. After both threads complete, the final value of count is printed. Since each thread increments the counter 1000 times and there are two threads, the expected value of count after both threads complete is: **1000 (increments by t1) +1000 (increments by t2) = 2000**

Communication between Threads

Communication between threads is crucial in multithreaded programming to ensure proper synchronization and data exchange. Java provides several ways to achieve inter-thread communication, including **wait()**, **notify()**, **notifyAll()**, and other concurrency utilities from the **java.util.concurrent** package. Here, we are going to discuss inter-thread communication using the **wait()** and **notify()** methods.

To facilitate communication among threads, the **wait()**, **notify()**, and **notifyAll()** methods are used. These methods must be invoked within a synchronized method or synchronized block on the object that is calling them; otherwise, an **IllegalMonitorStateException** will be thrown.

The **wait()** method causes a thread to wait until a specific condition is met. When this condition occurs, the **notify()** or **notifyAll()** methods can be called to alert waiting threads to resume their normal execution. The **notifyAll()** method wakes up all waiting threads, while the **notify()** method wakes up only one thread from the waiting queue.

A classic example of communication between threads is the Producer-Consumer Problem. One thread (the producer) produces data, and another thread (the consumer) consumes it. The producer and consumer must be synchronized to avoid conflicts.

```
1. class SharedResource {
2.     private int data;
3.     private boolean dataAvailable = false;
4.
5.     public synchronized void produceData(int value) {
6.         while (dataAvailable) {
```

```
            try {
                wait();  // Wait until the data is consumed
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        data = value;
        dataAvailable = true;
        System.out.println("Produced: " + value);
        notify();  // Notify the consumer that data is available
    }

    public synchronized void consumeData() {
        while (!dataAvailable) {
            try {
                wait();  // Wait until data is produced
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        System.out.println("Consumed: " + data);
        dataAvailable = false;
        notify();  // Notify the producer that the data has been
consumed
    }
}

class ProducerThread extends Thread {
    private SharedResource resource;

    public ProducerThread(SharedResource resource) {
        this.resource = resource;
    }

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            resource.produceData(i);
            try {
                Thread.sleep(100);  // Simulate time taken to produce
```

```
data
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}

class ConsumerThread extends Thread {
    private SharedResource resource;

    public ConsumerThread(SharedResource resource) {
        this.resource = resource;
    }

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            resource.consumeData();
            try {
                Thread.sleep(150);  // Simulate time taken to
consume data
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}

public class Main {
    public static void main(String[] args) {
        SharedResource resource = new SharedResource();
        ProducerThread producer = new
ProducerThread(resource);
        ConsumerThread consumer = new
ConsumerThread(resource);

        producer.start();
        consumer.start();
    }
}
```

SharedResource class contains the shared data and methods for producing and consuming data. It uses synchronized methods to ensure mutual exclusion and **wait()/notify()** for coordination. **produceData(int value)** produces data if no data is currently available. Otherwise, it waits. **consumeData()** consumes data if data is available. Otherwise, it waits.

ProducerThread class extends Thread and repeatedly produces data by calling the **produceData()** method of **SharedResource**. **ConsumerThread** class extends Thread and repeatedly consumes data by calling the **consumeData()** method of **SharedResource**. Main class creates instances of **ProducerThread** and **ConsumerThread** and starts their execution. The synchronized keyword ensures that only one thread can access the **produceData()** or **consumeData()** method at a time, preventing data inconsistency. **wait()** causes the current thread to wait until another thread invokes **notify()** or **notifyAll()** for the same object. **notify()** wakes up a single thread that is waiting on the object's monitor. Thread **Sleep()** is used to simulate the time taken for producing and consuming data, making the output more readable and realistic. The program will give the following output:

Produced: 0
Consumed: 0
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Consumed: 3
Produced: 4
Consumed: 4

Real-world examples

The following are some real-world examples:

- **Bank account management**: Managing multiple accounts and performing transactions concurrently.

```
1. class Account {
2.     private int balance = 1000;
```

```
3.
4.      public synchronized void depositAmount(int amount) {
5.          balance += amount;
6.          System.out.println("Deposited " + amount + ", New
    Balance: " + balance);
7.      }
8.
9.      public synchronized void withdrawAmount(int amount)
    {
10.         if (balance >= amount) {
11.             balance -= amount;
12.             System.out.println("Withdrew " + amount + ",
    New Balance: " + balance);
13.         } else {
14.             System.out.println("Insufficient funds to withdraw "
    + amount);
15.         }
16.     }
17. }
18.
19. class Transaction extends Thread {
20.     private Account account;
21.     private boolean isDeposit;
22.
23.     public Transaction(Account account, boolean isDeposit)
    {
24.         this.account = account;
25.         this.isDeposit = isDeposit;
26.     }
27.
28.     public void run() {
29.         if (isDeposit) {
30.             account.depositAmount(100);
31.         } else {
32.             account.withdrawAmount(100);
33.         }
34.     }
35. }
36.
37. public class Main {
38.     public static void main(String[] args) {
```

```
        Account account = new Account();
        Thread depositThread1 = new Transaction(account, true); // Deposit
        Thread withdrawThread1 = new Transaction(account, false); // Withdraw
        Thread depositThread2 = new Transaction(account, true); // Deposit
        Thread withdrawThread2 = new Transaction(account, false); // Withdraw

        depositThread1.start();
        withdrawThread1.start();
        depositThread2.start();
        withdrawThread2.start();

        try {
            depositThread1.join();
            withdrawThread1.join();
            depositThread2.join();
            withdrawThread2.join();
        } catch (InterruptedException e) {
            System.out.println(e);
        }
    }
}
```

Account class handles deposits and withdrawals synchronously to avoid race conditions. **Transaction** class performs transactions concurrently. Synchronization ensures that account balance updates are handled correctly.

- **Real-time stock price monitoring:**

  Scenario: Simulating a stock price monitor that updates and displays stock prices in real-time.

```
import java.util.Random;

class StockPriceMonitor extends Thread {
    private String stockName;

    public StockPriceMonitor(String stockName) {
        this.stockName = stockName;
```

```
    }

    public void run() {
        Random random = new Random();
        while (true) {
            double price = 100 + random.nextDouble() * 50; // Simulate a stock price
            System.out.println(stockName + " Current Price: $" + String.format("%.2f", price));
            try {
                Thread.sleep(1000); // Update every second
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}

public class StockMonitorApp {
    public static void main(String[] args) {
        Thread stock1 = new StockPriceMonitor("AAPL");
        Thread stock2 = new StockPriceMonitor("GOOGL");

        stock1.start();
        stock2.start();
    }
}
```

StockPriceMonitor simulates real-time stock price. Updates.Random generates random stock prices. Thread.sleep(1000) updates prices every second.

Conclusion

In this chapter, we explored the fundamentals of concurrent programming and multithreading in Java. We began by understanding the concept of multithreading, which allows for the concurrent execution of multiple parts of a program, thereby maximizing CPU utilization and improving application performance. Through various examples and code snippets, we delved into the creation and management of threads using the

Thread class and the Runnable interface.

We covered the life cycle of a thread, from its creation to termination, and discussed the different states a thread can be in during its execution. We also learned about the significance of thread synchronization and how to use synchronized methods and blocks to prevent thread interference and ensure data consistency when multiple threads access shared resources.

The chapter provided practical insights into the use of key thread methods such as start(), run(), sleep(), join(), and interrupt(). These methods are essential for controlling thread behavior and coordinating their execution. We also examined best practices for implementing these methods to develop efficient and reliable multithreaded applications.

In the next chapter, we will learn GUI programming in Java.

Exercises

1. Write a Java program that creates a thread to print "Hello, World!" and then terminates.
2. Write a Java program that creates a thread to count from 1 to 10, with each number printed after a 1-second delay.
3. Write a Java program that creates three threads with different priorities (MIN_PRIORITY, NORM_PRIORITY, MAX_PRIORITY) and observe their execution order.
4. Write a Java program that creates a thread using the Runnable interface to print the numbers from 1 to 5.
5. Write a Java program that creates two threads: one to print even numbers from 1 to 10 and another to print odd numbers from 1 to 10.
6. Write a Java program that creates two threads to increment a shared counter 1000 times each. Use synchronization to ensure thread safety.

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 9

Swing GUI Fundamentals

Introduction

Swing is used for building **graphical user interfaces (GUIs)** in Java applications. Being a crucial part of the **Java Foundation Classes (JFC)**, introduced as a more advanced and flexible successor to the **Abstract Window Toolkit (AWT)**. Swing provides a rich set of lightweight components that can be easily customized and extended. Unlike AWT, which relies heavily on the native system's GUI controls, Swing components are written entirely in Java, allowing for a more consistent look and feel across different platforms. This toolkit includes everything from basic components like buttons, labels, and text fields to complex components such as tables, trees, and text editors. Additionally, Swing supports a pluggable look-and-feel architecture, which enables developers to create applications with a variety of appearances and behaviors. All the classes and interfaces of the Swing library are kept inside javax.swing package. With its extensive component library, powerful event handling capabilities, and ease of use, Swing has become a popular choice for creating robust, platform-independent

GUIs in Java.

Structure

The chapter covers the following topics:

- Containers
- Components
- Layout managers in containers
- Event handling

Objectives

The objective of this chapter is to introduce the fundamental concepts and components of Java Swing, a GUI toolkit used for building rich desktop applications. By the end of the chapter, readers will have a thorough understanding of different types of containers, including top-level and intermediate-level containers, and their roles in the Swing architecture, Core Swing components such as JLabel, ImageIcon, JButton, JTextField, JList, and JComboBox, which are essential for creating user interfaces. Readers will also be able to understand the importance of layout managers in organizing and arranging GUI components within containers. The chapter also introduces event-driven programming model in Swing, focusing on how user interactions trigger events. Furthermore, an exposure to adapter classes and how they simplify the implementation of event listeners.

By mastering these topics, readers will be able to design and implement user-friendly graphical interfaces in Java, enhancing their ability to create interactive and professional desktop applications.

Containers

In Java Swing, containers are fundamental components that hold and manage the layout of other components. They are used to organize and arrange the GUI elements like buttons, labels, text fields, and other components. All the components are added to containers to make them visible. Containers can also be nested,

meaning a container can hold other containers, which in turn hold other components.

Containers in Swing are classified into two main categories:

- Top-level containers
- Intermediate (or general-purpose) containers

Top-level containers

Top-level containers are the highest-level containers in Swing. They provide the basic window structure, and every Swing application must have at least one top-level container. Common top-level containers are **JFrame, JDialog, JApplet and JWindow**. Each top-level container in Swing has a hierarchical structure composed of multiple layers. Understanding this structure is crucial for effectively managing components and layouts within the container. Layers of top-level containers are as follows:

- **Root pane (JRootPane)**: Every top-level container has a JRootPane, which serves as the main interface between the top-level container and its components. The JRootPane manages the content pane, layered pane, glass pane, and optional menu bar.
- **Layered pane (JLayeredPane)**: The JLayeredPane allows components to be layered on top of each other within the container. This enables features like pop-up windows and tooltips, where one component can appear above another.
- **Content pane (JContentPane)**: The content pane is where most components (like buttons, panels, text fields) are added. This is the main area of the container that the application directly interacts with.
- **Glass pane (JGlassPane)**: The glass pane is a transparent pane that sits over the entire container. It is often used for intercepting mouse events or drawing custom decorations over the window.
- **Optional menu bar (JMenuBar)**: If the container is a JFrame or JDialog, it can also contain a menu bar, which provides a place for application menus (e.g., File, Edit, Help).

The structure of a top-level container class is shown below:

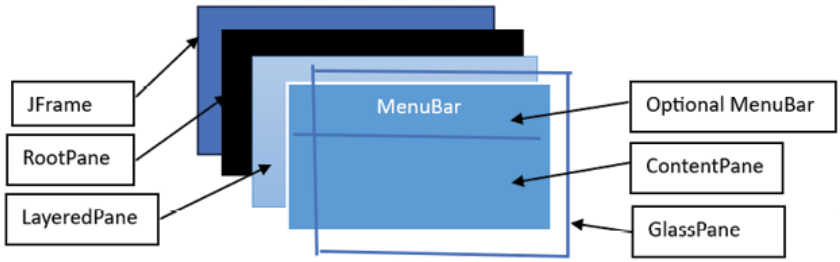


***Figure 9.1**: Structure of top-level container*

JFrame

JFrame is a fundamental class in Java's Swing library, used to create the main window for a GUI application. As a top-level container, JFrame serves as the primary window where all other components, such as buttons, text fields, panels, and menus are added. It provides standard window decorations with a title bar, borders, and buttons to minimize, maximize, or close the window. The class hierarchy of JFrame is as follows:

- `java.lang.Object`
- `java.awt.Component`
- `java.awt.Container`
- `java.awt.Window`
- `java.awt.Frame`
- `javax.swing.JFrame`

***Figure 9.2**: Class hierarchy of JFrame*

All components in a JFrame are added to its content pane. By default, the content pane uses a BorderLayout, but you can set a different layout manager. JFrame supports adding a JMenuBar for creating application menus. The menu bar is placed at the top of the window, below the title bar. By default, a JFrame is resizable, meaning the user can adjust the window size. However, you can disable resizing by calling **setResizable(false)**. JFrame supports

various event listeners, such as WindowListener for handling window events (e.g., closing, opening, minimizing), ComponentListener for handling component events, and KeyListener for handling keyboard input. JFrame provides options for handling what happens when the user closes the window. You can set the default close operation to exit the application, dispose of the frame, or do nothing.

Commonly used constructors of JFrame class are:

- **JFrame()**: Creates a new, initially invisible JFrame with no title.
- **JFrame(String title)**: Creates a new, initially invisible JFrame with the specified title.

The following are the commonly used methods:

Method	Description
void setTitle(String title)	Sets the title of the frame.
String getTitle()	Returns the title of the JFrame.
void setSize(int width, int height)	Sets the size of the JFrame.
void setBounds(int x, int y, int width, int height)	Sets the position and size of the JFrame.
void setLocation(int x, int y)	Sets the location of the JFrame on the screen.
void setVisible(boolean b)	Makes the JFrame visible or invisible.
void setDefaultCloseOperation(int operation)	Sets the default close operation (e.g., EXIT_ON_CLOSE, DISPOSE_ON_CLOSE).
Container getContentPane()	Returns the content pane of the JFrame where components are added.
void setContentPane(Container contentPane)	Sets the content pane of the JFrame.
void add(Component comp)	Adds a component to the content pane of the JFrame.
void setJMenuBar(JMenuBar menubar)	Sets the menu bar for the JFrame.
void setResizable(boolean resizable)	Sets whether the JFrame is resizable by the user.
void setIconImage(Image image)	Sets the icon image for the JFrame, which appears in the title bar.
void setExtendedState(int state)	Sets the extended state of the JFrame (e.g., MAXIMIZED_BOTH, ICONIFIED).
void dispose()	Releases the resources used by the JFrame and closes it.

***Table 9.1**: commonly used methods of JFrame*

Here is an example demonstrating how to create and use a JFrame:

```
1. import javax.swing.*;
2. public class JFrameExample {
3.     public static void main(String[] args) {
4.         JFrame frame = new JFrame("My Application");
5.         frame.setSize(400, 300);
6.         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
7.         frame.setVisible(true);
8.     }
```

```
9. }
```

The provided code is a basic example of creating a simple GUI using Java Swing. The code defines a class JFrameExample with a main method that sets up a JFrame, which is a top-level container used to represent a window in a Swing application. The JFrame is titled "**My Application**" and is set to a size of *400x300* pixels. The **setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)** method ensures that the application will exit when the user closes the window. Finally, **setVisible(true)** makes the frame visible on the screen. JFrame class has various features. Let us discuss them in detail:

- **Content pane and layout managers**: The JFrame's content pane is where you add components like buttons, labels, and panels. The layout manager of the content pane controls how these components are arranged.
- **Event handling**: Handling events like closing the window or clicking a button is essential in GUI applications. For example, setting the default close operation to EXIT_ON_CLOSE ensures that the application terminates when the window is closed.
- **Customizing the JFrame**: JFrame allows for various customizations, such as setting an icon image that appears in the title bar or changing the window's title. You can also control whether the window is resizable or maximized by default.
- **Nested components**: JFrame can contain other containers, such as JPanel, which in turn can contain multiple components. This allows for complex layouts and a structured, organized GUI.
- **Visibility and size**: Controlling the visibility and size of a JFrame is crucial for creating user-friendly interfaces. You can set the window size explicitly or allow the window to pack itself based on the preferred sizes of its components using **pack()**.
- **Look and feel**: JFrame, like all Swing components, follows the "**Look and Feel**" principle, allowing you to change the appearance of the entire application to match different

platforms or custom styles.

JWindow

JWindow is a simple top-level container that does not have a title bar, menu bar, or borders. This undecorated nature allows for a cleaner, more flexible presentation, especially when standard window elements are not needed. As a top-level container, JWindow does not have a parent container. It is self-contained and can serve as a root container for other Swing components. JWindow is often used for lightweight, temporary windows, such as splash screens, where you need a window that appears on top of other windows but does not need user interaction. Unlike JFrame, JWindow does not have a close button, so it must be programmatically closed using the **dispose()** method or by calling **setVisible(false)**. JWindow allows for transparency and shaped windows, making it suitable for custom UI designs that require non-rectangular windows or see-through effects. JWindow class extends from **java.awt.Window** class, which extends from **java.awt.Container**.

The following are the commonly used constructors:

- **JWindow()**: Creates a new, initially invisible JWindow with no specified owner.
- **JWindow(Window owner)**: Creates a new JWindow with the specified owner window. The JWindow will be attached to the owner, meaning it will always appear above it and will be disposed of when the owner is closed.

The following are the commonly used methods:

Method	Description
void setVisible(boolean b)	Makes the JWindow visible or invisible.
void dispose()	Releases all resources used by the JWindow and hides it.
void setSize(int width, int height)	Sets the size of the JWindow.
void setLocation(int x, int y)	Sets the location of the JWindow on the screen.
void setContentPane(Container contentPane)	Sets the content pane of the JWindow, where components are added.
void setBackground(Color color)	Sets the background color of the JWindow.
void setOpacity(float opacity)	Sets the opacity of the JWindow, allowing for transparency.
void setShape(Shape shape)	Sets the shape of the JWindow, enabling non-rectangular window shapes.

***Table 9.2**: Commonly used methods of JWindow class*

Here is an example code of creating a simple splash screen using JWindow:

```
// Create a JWindow instance
JWindow window = new JWindow();

// Set the size and location of the JWindow
window.setSize(300, 200);
window.setLocationRelativeTo(null);
// Center the window on the screen
// Create a label to display inside the JWindow
JLabel label = new JLabel("Loading, please wait...",
JLabel.CENTER);
label.setFont(new Font("Arial", Font.BOLD, 16));
// Set the content pane of the JWindow
window.getContentPane().add(label);
// Set the background color and opacity (optional)
window.setBackground(new Color(0, 0, 0, 0));
// Fully transparent background
window.setOpacity(0.85f); // Slightly transparent window
// Make the JWindow visible
window.setVisible(true);
// Simulate a loading process
try {
   Thread.sleep(3000); // Display the splash screen for 3 seconds
} catch (InterruptedException e) {
   e.printStackTrace();
}
```

Complete program is available on GitHub link. Here, the JWindow is created without an owner, making it a standalone window. It is then set to a specific size and centered on the screen. A JLabel is added to the content pane of the JWindow, displaying a message to the user. The label is centered within the window. The background is set to be fully transparent, and the opacity is set to 85%, giving the window a semi-transparent look. The window is made visible, simulating a loading process for 3 seconds, after which the JWindow is disposed of, freeing resources and closing the window.

Some use cases for JWindow are given below:

- **Splash screens**: JWindow is ideal for splash screens that display while the main application is loading. The lack of window decorations allows for a cleaner, more polished splash screen.
- **Custom-shaped windows**: JWindow supports custom window shapes, making it suitable for applications that require non-rectangular windows, such as creative design tools or games.
- **Pop-up notifications**: For temporary notifications that appear above all other windows, JWindow provides a simple, undecorated option that can be shown and hidden as needed.
- **Tooltips and floating panels**: JWindow can be used for tooltips or floating panels that require a lightweight and flexible window without the overhead of a full JFrame.

JDialog

The JDialog class in Java is a part of the Swing library, which is used to create dialog boxes in GUIs. JDialog is a top-level container that can be used to create custom dialog windows. These dialogs are typically used for user interaction, such as displaying messages, gathering input, or confirming actions. Like JFrame, JDialog is a top-level container, meaning it can exist independently of other containers, although it often works in conjunction with a parent JFrame.

JDialog supports modality, which means it can block user input to other windows in the application until the dialog is dismissed. There are several types of modality:

- **Modeless**: The dialog does not block input to other windows.
- **Application modal**: The dialog blocks input to all windows of the application.
- **Document modal**: The dialog blocks input to all windows from the same document.
- **Toolkit modal**: The dialog blocks input to all windows

across applications in the same toolkit.

You can add any component to a JDialog, such as buttons, text fields, labels, or even panels. This makes it highly customizable for different types of dialogs. A JDialog is typically associated with a parent JFrame or another JDialog. The dialog window is centered relative to its parent when it appears, and when the parent is closed, the dialog is automatically disposed of.

The following are the common constructors:

- **JDialog()**: Creates a modeless dialog with no parent.
- **JDialog(Frame owner)**: Creates a modeless dialog with the specified frame as its owner.
- **JDialog(Frame owner, boolean modal)**: Creates a dialog with the specified frame as its owner and with the specified modality.
- **JDialog(Frame owner, String title, boolean modal)**: Creates a dialog with the specified frame as its owner, with the specified title, and with the specified modality.

The following are the common methods of JDialog:

- **Setting modality**: **setModal(boolean modal)** sets whether the dialog is modal.
- **Setting title**: **setTitle(String title)** sets the title of the dialog.
- **Adding components**: **add(Component comp)** adds a component to the dialog's content pane.
- **Display dialog**: **setVisible(boolean b)** makes the dialog visible or invisible.
- **Disposing dialog**: **dispose()** releases all of the native screen resources used by this dialog, its subcomponents, and all of its owned children. You can see an example program of JDialog on the GitHub link.

JApplet

JApplet is a container used to create applets, which are small Java applications that run in a web browser or applet viewer. With the decline of applets, JApplet is rarely used in modern Java

applications. JApplet class extends from **java.applet.Applet** class, which extends from **java.awt.Panel**. Applet has its own lifecycle. Its Lifecycle Methods are as follows:

- **init()**: Called when the applet is first loaded. This is where you typically set up the GUI, initialize variables, and configure your applet.
- **start()**: Called after **init()** and whenever the applet is restarted. This method is used to start or resume execution, such as starting animations or threads.
- **stop()**: Called when the applet is stopped, usually when the user navigates away from the page. It is used to suspend execution, such as stopping animations or threads.
- **destroy()**: Called when the applet is being removed from memory. This is where you would clean up resources.

JApplet allows you to create a windowed interface within a web browser. Applets have some limitations. Applets have limited access to the user's system due to security restrictions, especially in web browsers. They cannot easily access files on the user's computer, connect to arbitrary servers, or execute local programs. The below mentioned program demonstrates the usage of **JApplet** class:

```
1. import javax.swing.*;
2. import java.awt.event.ActionEvent;
3. import java.awt.event.ActionListener;
4. public class MyJApplet extends JApplet {
5.     private JButton button;
6.     private JLabel label;
7.     @Override
8.     public void init() {
9.         // Set up GUI components
10.        button = new JButton("Click Me");
11.        label = new JLabel("Hello, JApplet!");
12.        // Add components to content pane
13.        getContentPane().add(button, "North");
14.        getContentPane().add(label, "Center");
15.        // Set up event handling
16.    }
17.    @Override
18.    public void start() {
```

```
19.         // Code to start or resume applet (e.g., start animation)
20.     }
21.     @Override
22.     public void stop() {
23.         // Code to stop applet (e.g., stop animation)
24.     }
25.     @Override
26.     public void destroy() {
27.         // Clean up resources
28.     }
29. }
```

To execute a JApplet in Java compile your **MyJApplet.java** file using the **javac** command. Create an HTML file that references your applet. For example, **MyJApplet.html**:

```
<html>
<body>
    <applet code="MyJApplet.class" width="300" height="300">
    </applet>
</body>
</html>
```

Applets are typically run in a web browser or an applet viewer. To run your applet using the Applet Viewer, use the command:

appletviewer MyJApplet.html

This will open the applet viewer and execute your **MyJApplet** class, displaying the GUI components defined in the **init()** method.

Please note that the use of JApplet and applets, in general, has become outdated, as modern browsers no longer support Java applets. If you are working with Java GUI applications, consider using JFrame or other Swing components instead.

Intermediate or general-purpose containers

Intermediate containers are used to group components within top-level containers. They help manage the layout and organization of components inside the top-level containers. Common intermediate containers are JPanel, JScrollPane, JTabbedPane, JSplitPane etc. Let us understand these containers in detail.

JPanel

JPanel is the most commonly used intermediate container. It is a simple, lightweight container that is often used to group related components. A JPanel can also be nested within other containers, making it a versatile tool for organizing complex GUIs. The syntax of creating and using panel is as follows:

```
JPanel panel = new JPanel(); // Create a JPanel
// Add some components to the panel
panel.add(new JButton("Button 1"));
```

JScrollPane

The JScrollPane class in Java Swing is a component that provides a scrollable view of another component. It is particularly useful when you have content that might not fit within the visible area of the window or container, such as long lists, large images, or extensive text. For example, it is commonly used to make lists, tables, or text areas scrollable when their content exceeds the visible area.

JScrollPane automatically provides horizontal and/or vertical scroll bars when the content inside it exceeds the viewable area. The scroll bars appear as needed, and they allow the user to scroll through the content either by dragging the scroll thumb, clicking on the scroll arrows, or using the scroll wheel. The content inside a JScrollPane is managed by a JViewport, which represents the visible portion of the content. The viewport manages which part of the content is visible and updates as the user scrolls. You can access the viewport using the **getViewport()** method. While JScrollPane can be used with any component, it is most commonly used with components that implement the Scrollable interface, like JTextArea, JTable, JList, and JTree. These components have built-in support for scrolling. However, you can make any component scrollable by placing it inside a JScrollPane.

You can control when the scroll bars appear using scroll bar policies. Custom scroll bar policies are:

- JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS
- JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED (default)

- JScrollPane.HORIZONTAL_SCROLLBAR_NEVER
- JScrollPane.VERTICAL_SCROLLBAR_ALWAYS
- JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED (default)
- JScrollPane.VERTICAL_SCROLLBAR_NEVER

These policies determine whether the scroll bars are always visible, visible only when necessary, or never visible.

You can customize the scroll bars (e.g., change their appearance or behavior) by accessing them through **getHorizontalScrollBar()** and **getVerticalScrollBar()**.

The following code snippet shows changing the unit increment (how much the scroll bar moves with each click):

```
JScrollPane scrollPane = new JScrollPane(component);
scrollPane.getVerticalScrollBar().setUnitIncrement(16);
```

A simple example of using **JScrollPane** with a JTextArea is provided on the GitHub link.

Common use cases of using JScrollbar are:

- **Large text areas**: Wrapping a JTextArea inside a JScrollPane allows users to scroll through large amounts of text.
- **Tables and lists**: JTable and JList are often placed inside a JScrollPane to handle large datasets that extend beyond the visible area.
- **Custom components**: Any custom component that has content larger than its display area can benefit from being wrapped in a JScrollPane.

JTabbedPane

The **JTabbedPane** class in Java Swing provides a mechanism for creating a tabbed interface, where multiple panels (or components) are contained within different tabs. This allows users to switch between different views or sets of components by clicking on the tabs. This is useful for organizing content into categories that the user can easily switch between.

You can add a tab to **JTabbedPane** using the **addTab**(String title, Component component) method. The title appears on the tab, and

the component is what is displayed when the tab is selected. You can programmatically select a tab using **setSelectedIndex(int index)** or **setSelectedComponent(Component component)**. You can listen for tab selection changes using a ChangeListener added to the **JTabbedPane**. This allows you to take action when a user selects a different tab. The following code snippet shows creating and using **JTabbdPane**:

```
// Create a JTabbedPane
JTabbedPane tabbedPane = new JTabbedPane();
// Add tabs to the JTabbedPane
tabbedPane.addTab("Tab 1", panel1);
```

A simple example demonstrating the use of **JTabbedPane** is provided on the GitHub link.

The following are the common use cases for **JTabbedPane**:

- **Settings panels**: JTabbedPane is commonly used in applications where different settings or configuration panels need to be grouped together.
- **Multi-document interfaces**: It can be used to implement a multi-document interface (MDI) where each tab represents an open document or file.
- **Dashboard views**: In applications that have multiple dashboard views or reports, each can be placed in a separate tab for easy navigation.

JSplitPane

The **JSplitPane** class in Java Swing is a container that divides two components (or panes) with a draggable divider, allowing the user to adjust the size of each pane interactively. It's particularly useful for applications where you need to display two related pieces of content side by side or one above the other, with the ability to resize them dynamically. JSplitPane divides its area into two sections, each of which can contain any component (like JPanel, JScrollPane, etc.). JSplitPane can be oriented either horizontally or vertically:

- **Horizontal (JSplitPane.HORIZONTAL_SPLIT)**: The two components are placed side by side, and the divider runs vertically between them.

- **Vertical (JSplitPane.VERTICAL_SPLIT)**: The two components are stacked one above the other, and the divider runs horizontally between them.

The divider is the draggable bar between the two panes. It can be customized in terms of appearance, size, and position. You can set the initial location of the divider using **setDividerLocation(int location)** and change its size using **setDividerSize(int newSize)**. The following code snippet shows creating and using JSplitPane:

```
// Create a JSplitPane with horizontal split
JSplitPane splitPane = new
JSplitPane(JSplitPane.HORIZONTAL_SPLIT, leftPanel, rightPanel);
// Set some properties
  splitPane.setDividerLocation(150); // Initial divider location
  splitPane.setOneTouchExpandable(true); // Allow one-touch
expansion
  splitPane.setDividerSize(10); // Set the size of the divider
```

A simple example demonstrating the use of JSplitPane is shared on the GitHub link.

Components

Swing components are the building blocks of a graphical user interface in Java. These components provide a rich set of controls and containers that allow developers to create complex and user-friendly applications. When discussing Swing components in Java, it is important to explain the JComponent class because it serves as the foundation for all Swing components. Every Swing component, such as JButton, JLabel, JPanel, etc., inherits from JComponent. This means that all these components share the same core set of methods and properties provided by JComponent.

JComponent

The JComponent class is the base class for all Swing components except top-level containers such as JFrame, JDialog, and JApplet. It provides the common functionality needed by all lightweight Swing components, such as painting, event handling, layout management, and accessibility support. The Class Hierarchy of JComponent class is as follows:

- java.lang.Object
- java.awt.Component
- java.awt.Container
- javax.swing.JComponent

***Figure 9.3**: Class hierarchy of JComponent*

Some of its commonly used methods are:

Method	Description
`void setBorder(Border border)`	Sets the border of the component.
`Border getBorder()`	Gets the border of the component.
`void setToolTipText(String text)`	Sets the tooltip text that appears when the user hovers over the component.
`String getToolTipText()`	Gets the tooltip text set for the component.
`void setVisible(boolean visible)`	Sets the component's visibility.
`boolean isVisible()`	Returns true if the component is visible, otherwise false.
`void setEnabled(boolean enabled)`	Enables or disables the component. A disabled component cannot be interacted with.
`boolean isEnabled()`	Returns true if the component is enabled, otherwise false.
`void addMouseListener(MouseListener l)`	Adds a mouse listener to handle mouse events.
`void setBackground(Color bg)`	Sets the background color of the component.
`Color getBackground()`	Gets the background color of the component.
`void setForeground(Color fg)`	Sets the foreground color of the component (e.g., text color).
`Color getForeground()`	Gets the foreground color of the component.
`void setFont(Font font)`	Sets the font used by the component.
`Font getFont()`	Gets the font used by the component.

***Table 9.3**: Commonly used methods of JComponent class*

JLabel

JLabel is a fundamental Swing component used to display a short string or an image icon or both. It is a non-editable text component, meaning users cannot modify the text displayed by JLabel. It is typically used for labeling other GUI components or providing informational text to the user. The text and icon can be aligned horizontally and vertically within the label.

JLabel has several constructors to create a label with or without text and icons:

- **Default constructor**:
  JLabel label = new JLabel();
  Creates an empty label.
- **Constructor with text**:
  JLabel label = new JLabel("Hello, World!");
  Creates a label with specified text.
- **Constructor with icon**:
  Icon icon = new ImageIcon("path/to/image.png");
  JLabel label = new JLabel(icon);
  Creates a label with an icon.
- **Constructor with text and icon**:
  Icon icon = new ImageIcon("path/to/image.png");
  JLabel label = new JLabel("Hello, World!", icon, JLabel.CENTER);
  Creates a label with specific text and icon, with the icon positioned relative to the text.

Here are some common methods used with JLabel:

	Description	

***Table 9.4**: List of common methods of JLabel class*

ImageIcon

ImageIcon is a class in the Java Swing library that represents an icon made from an image. It is commonly used with Swing components like JLabel, JButton, JMenuItem, and others to display images. It can load images from a file, URL, or array of image pixels. It can be resized to fit specific dimensions. Provides smooth rendering of images in various components.

ImageIcon provides several constructors to create icons from different sources:

- **Default constructor**:

  ImageIcon icon = new ImageIcon();

  Creates an empty ImageIcon.

- **Constructor with File Path**:

  ImageIcon icon = new ImageIcon("path/to/image.png");

  Creates an ImageIcon from an image file.

- **Constructor with URL**:

  ```
  try {
      URL url = new URL("http://example.com/image.png");
      ImageIcon icon = new ImageIcon(url);
  } catch (MalformedURLException e) {
      e.printStackTrace();
  }
  ```

  Creates an ImageIcon from a URL.

Here are some common methods used with ImageIcon:

	Description	
getImage()	icon.getImage();	this icon.
setImage(Image image)	Toolkit.getDefaultToolkit().getImage("path/to/newImage.png"); icon.setImage(newImage);	

__Table 9.5__: Commonly used methods with ImageIcon class

Buttons

Buttons are a fundamental part of the Java Swing GUI toolkit, used to perform actions when clicked by the user. All button classes extend from javax.swing.AbstractButton class, which intern extends from Javax.swing.JComponent class. Here is a detailed description of the various types of buttons in Swing and their key features:

JButton

The JButton class in Swing is used to create a button in a GUI application in Java. The JButton is a basic component that triggers an action when the user clicks on it. A JButton can display text, an icon, or both. You can create a button with just text, just an icon, or

both by passing the appropriate parameters to the JButton constructor. When the button is clicked, it triggers an ActionEvent, which can be handled using an ActionListener. You can add an ActionListener to a JButton using the addActionListener method. The appearance of the button can be customized using methods inherited from JComponent and AbstractButton. This includes setting the background, foreground, font, and border.

The following are the constructors:

- **JButton()**: Creates a button with no text or icon.
- **JButton(String text)**: Creates a button with specific text.
- **JButton(Icon icon)**: Creates a button with a specified icon.
- **JButton(String text, Icon icon)**: Creates a button with both text and an icon.

The following are the methods of JButton. Some of the commonly used methods of JButton include:

- **setText(String text)**: Sets the text of the button.
- **getText()**: Returns the text of the button.
- **setIcon(Icon icon)**: Sets the icon of the button.
- **getIcon()**: Returns the icon of the button.
- **setEnabled(boolean enabled)**: Enables or disables the button.
- **isEnabled()**: Returns the enabled status of the button.
- **addActionListener(ActionListener l):** Adds an action listener to the button.
- **removeActionListener(ActionListener l)**: Removes an action listener from the button.
- **setToolTipText(String text):** Sets the tooltip text of the button.

  Allows for customization of appearance using **setBackground()**, **setForeground()**, **setFont()**, etc.

Let us see one example program for creating and using a JButton:

```
1. import javax.swing.JButton;
2. import javax.swing.JFrame;
3. import java.awt.event.ActionEvent;
```

```
import java.awt.event.ActionListener;
public class JButtonExample {
    public static void main(String[] args) {
        // Create a new JFrame container
        JFrame frame = new JFrame("JButton Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(null);

        // Create a new JButton
        JButton button = new JButton("Click Me");
        button.setBounds(100, 50, 100, 30);

        // Add an ActionListener to the button

        // Add the button to the frame
        frame.add(button);

        // Make the frame visible
        frame.setVisible(true);
    }
}

```

The JButtonExample Java Swing application creates a simple graphical user interface featuring a single button within a JFrame. The frame is initialized with a size of *300x200* pixels and uses absolute positioning (with **setLayout(null)**). To place the button precisely at coordinates (100, 50) with a width of 100 pixels and a height of 30 pixels **setBounds()** method was used. The setBounds method is used to set the size and position of a component within its container. This method is part of the **Component** class, which is a superclass of all Swing components. It allows you to specify the location and size of the component using a single call. An **ActionListener** should be added to the button to handle user interaction. The frame is made visible to the user, showcasing how to set up basic Swing components, manage layout manually, and respond to user actions.

JToggleButton

JToggleButton is a Swing component that represents a button which can be toggled on or off. Unlike a regular JButton, which is either pressed or not, a JToggleButton maintains a state—either selected (on) or unselected (off). This makes it useful in scenarios where you need a button that behaves like a switch. The primary feature of JToggleButton is its ability to maintain a selected state. When clicked, it toggles between selected and unselected states. You can check the state using the **isSelected()** method and change it using the **setSelected(boolean selected)** method. Like JButton, a JToggleButton can display text, an icon, or both. You can set these properties when creating the button or later using setter methods. JToggleButton triggers an ActionEvent when it is clicked, similar to JButton. You can add an ActionListener to handle these events. The appearance of JToggleButton can be customized using methods inherited from JComponent and AbstractButton. This includes setting the background, foreground, font, and border.

The following are the constructors:

- **JToggleButton()**: Creates a toggle button with no text or icon.
- **JToggleButton(String text)**: Creates a toggle button with specified text.
- **JToggleButton(Icon icon)**: Creates a toggle button with a specified icon.
- **JToggleButton(String text, Icon icon)**: Creates a toggle button with both text and an icon.
- **JToggleButton(Action a)**: Creates a toggle button where the properties are defined by the Action supplied.

The following are the methods of JToggleButton. Some of the commonly used methods of JToggleButton include:

- **setSelected(boolean selected)**: Sets the selected state of the button.
- **isSelected()**: Returns true if the button is selected, otherwise false.
- **getText()**: Returns the text of the button.
- **setText(String text)**: Sets the text of the button.

- **setIcon(Icon icon)**: Sets the icon of the button.
- **getIcon()**: Returns the icon of the button.
- **addActionListener(ActionListener l)**: Adds an action listener to the button.
- **removeActionListener(ActionListener l)**: Removes an action listener from the button.
- **setToolTipText(String text)**: Sets the tooltip text of the button.

A simple example demonstrating the usage of the toggle button is shared on the GitHub link.

JCheckBox

JCheckBox is a component in the Swing library used to create a checkable box with a label. It is a subclass of JToggleButton, meaning it inherits the behavior of a toggle button but is specifically designed to represent a binary choice—checked (selected) or unchecked (deselected). This state can be toggled by the user or programmatically.

JCheckBox is commonly used in forms, dialogs, and other user interfaces where multiple options can be selected independently.

A JCheckBox can display text, an icon, or both. The text is typically positioned next to the checkbox, but the icon and text alignment can be customized. JCheckBox generates action events when the user interacts with it (e.g., clicking on it to change its state). This allows developers to respond to changes in the checkbox's state. Besides action events, JCheckBox also generates item events (ItemEvent), which are particularly useful when you need to monitor the state change (selected or deselected) of the checkbox.

The following are the commonly used constructors:

- **JCheckBox()**: Creates an unselected checkbox with no text or icon.
- **JCheckBox(String text)**: Creates an unselected checkbox with the specified text.
- **JCheckBox(String text, boolean selected)**: Creates a checkbox with the specified text and selection state.

- **JCheckBox(String text, Icon icon)**: Creates an unselected checkbox with the specified text and icon.
- **JCheckBox(String text, Icon icon, boolean selected)**: Creates a checkbox with the specified text, icon, and selection state.

The following are the commonly used methods:

Method	Description
`void setSelected(boolean selected)`	Sets the checkbox's selected state.
`boolean isSelected()`	Returns true if the checkbox is selected, otherwise false.
`void setText(String text)`	Sets the text to be displayed next to the checkbox.
`String getText()`	Returns the text displayed next to the checkbox.
`void addActionListener(ActionListener l)`	Adds an ActionListener which is triggered when the state changes.
`void addItemListener(ItemListener l)`	Adds an ItemListener to handle item state changes (selected/deselected).
`void setIcon(Icon icon)`	Sets the icon displayed with the checkbox.
`Icon getIcon()`	Returns the icon displayed with the checkbox.
`void setMnemonic(int mnemonic)`	Sets the mnemonic key character for the checkbox via keyboard shortcut.
`void setEnabled(boolean enabled)`	Enables or disables the checkbox.
`boolean isEnabled()`	Returns true if the checkbox is enabled, otherwise false.
`void setHorizontalTextPosition(int textPosition)`	Sets the horizontal position of the text relative to the checkbox.
`void setVerticalTextPosition(int textPosition)`	Sets the vertical position of the text relative to the checkbox.

***Table 9.6**: Commonly used methods of JCheckBox*

The following code snippets demonstrate creating and using JCheckBox:

```
// Create checkboxes
JCheckBox checkBox1 = new JCheckBox("Option 1");
JCheckBox checkBox2 = new JCheckBox("Option 2", true);
// Create a panel and add checkboxes to it
JPanel panel = new JPanel();
panel.add(checkBox1);
panel.add(checkBox2);
```

The complete code is available on the GitHub link.

JRadioButton

JRadioButton is a component in the Swing library used to create a button that can be selected or deselected. It is a specialized version of JToggleButton, but unlike JCheckBox, radio buttons are designed to work in groups where only one button in the group can be

selected at a time. This behavior is typically used to present a set of mutually exclusive options to the user. When JRadioButton instances are grouped together (typically using a ButtonGroup), only one button in the group can be selected at any given time. This is ideal for scenarios where the user needs to select a single option from multiple choices. Like other buttons in Swing, a JRadioButton can display text, an icon, or both. The text typically appears next to the radio button. JRadioButton generates both action events and item events when the user interacts with it. These events can be used to respond to changes in the radio button's state (selected or deselected).

The following are the commonly used constructors:

- **JRadioButton()**: Creates an unselected radio button with no text or icon.
- **JRadioButton(String text)**: Creates an unselected radio button with the specified text.
- **JRadioButton(String text, boolean selected)**: Creates a radio button with the specified text and selection state.
- **JRadioButton(String text, Icon icon)**: Creates an unselected radio button with the specified text and icon.
- **JRadioButton(String text, Icon icon, boolean selected)**: Creates a radio button with the specified text, icon, and selection state.

The following are the commonly used methods:

Method	Description
void setSelected(boolean selected)	button.
boolean isSelected()	dio button is selected, otherwise false.
void setText(String text)	t to the radio button.
String getText()	displayed next to the radio button.
void addActionListener(ActionListener l)	n, which is triggered when the state changes.
void addItemListener(ItemListener l)	e changes (selected/deselected).
void setIcon(Icon icon)	with the radio button.
Icon getIcon()	n displayed with the radio button.
void setMnemonic(char mnemonic)	e radio button via keyboard shortcut.
void setEnabled(boolean enabled)	Enables or disables the radio button.
boolean isEnabled()	dio button is enabled, otherwise false.
void setHorizontalTextPosition(int textPosition)	e radio button.
void setVerticalTextPosition(int textPosition)	e radio button.

Table 9.7: Commonly used methods of JRadioButton

Using ButtonGroup

To ensure that only one JRadioButton in a set can be selected at a time, you typically add the buttons to a ButtonGroup. The ButtonGroup itself does not have a visual representation; it simply manages the selection state of its buttons.

For making radio buttons mutually exclusive, the steps will be as follows:

```
ButtonGroup group = new ButtonGroup();
JRadioButton radioButton1 = new JRadioButton("Option 1");
JRadioButton radioButton2 = new JRadioButton("Option 2");
JRadioButton radioButton3 = new JRadioButton("Option 3");

group.add(radioButton1);
group.add(radioButton2);
group.add(radioButton3);
.............
```

A simple example demonstrating how to use JRadioButton is shared on the GitHub link.

JTextField

JTextField is a Swing component used for allowing the user to input a single line of text. It is one of the most commonly used components in Java GUI applications for gathering user input, such as entering names, addresses, search queries, or other text-based data. JTextField class extends from **javax.swing.text.JTextComponent** class, which in turn extends from **javax.swing.JComponent**. The text displayed in a JTextField can be customized in terms of font, color, and alignment. Developers can set properties like text color (setForeground), background color (setBackground), font (setFont), and alignment (setHorizontalAlignment). JTextField can generate various events that developers can handle, such as ActionEvent, which is triggered when the user presses Enter. It also supports other events like focus and key events, allowing for more interactive behavior. The text within a JTextField can be programmatically manipulated. You can set, get, and select text, as well as control the caret position,

providing flexibility in how user input is handled. While JTextField itself does not perform input validation, it is commonly used in conjunction with other logic to validate the user's input. For example, a JTextField might be used to collect a phone number, and the input could be validated to ensure it contains only digits.

The following are the commonly used constructors:

- **JTextField()**: Creates an empty text field with a default width of 0 columns.
- **JTextField(String text)**: Creates a text field initialized with the specified text.
- **JTextField(int columns)**: Creates an empty text field with the specified number of columns. The columns determine the preferred width of the text field.
- **JTextField(String text, int columns)**: Creates a text field initialized with the specified text and a specified number of columns.

The following are the commonly used methods:

Method	Description
void setText(String text)	Sets the text of the text field.
String getText()	Returns the text contained in the text field.
void setColumns(int columns)	Sets the number of columns in the text field. This affects the preferred width.
int getColumns()	Returns the number of columns in the text field.
void addActionListener(ActionListener l)	Adds an ActionListener that is triggered when the user presses Enter.
void setEditable(boolean editable)	Sets whether the text field is editable. If false, the user cannot modify the text.
boolean isEditable()	Returns true if the text field is editable, otherwise false.
void setHorizontalAlignment(int align)	Sets the horizontal alignment of the text (e.g., JTextField.LEFT).
int getHorizontalAlignment()	Returns the horizontal alignment of the text.
void selectAll()	Selects all the text in the text field.
void setCaretPosition(int position)	Sets the position of the caret (blinking cursor).
int getCaretPosition()	Returns the position of the caret.
void setToolTipText(String text)	Sets the tooltip text that appears when the user hovers over the text field.

***Table 9.8**: Commonly used methods of JTextField*

The ActionListener associated with JTextField is particularly useful for scenarios where pressing Enter should trigger an action, such as submitting a form or searching for a term. JTextField provides comprehensive methods for manipulating the text it contains,

including setting and getting the text, selecting portions of the text, and controlling the caret position. The following code snippet demonstrates how to use JTextField:

```
// Create a JTextField with initial text and 20 columns
   JTextField textField = new JTextField("Enter your name", 20);
// Add ActionListener to handle when the user presses Enter
// add the text field to a panel
panel.add(textField);
```

JList

JList is a Swing component used to display a list of items to the user. It allows for both single and multiple selections from the list, making it a versatile component for presenting a list of choices or data items in a GUI. The items in a JList can be strings, objects, or any data type that you can convert to a string representation. JList class extends from **javax.swing.JComponent**.

JList supports different selection modes:

- **SINGLE_SELECTION**: Only one item can be selected at a time.
- **SINGLE_INTERVAL_SELECTION**: Multiple items can be selected, but they must be contiguous.
- **MULTIPLE_INTERVAL_SELECTION**: Any combination of items can be selected, including non-contiguous items.

JList uses a ListModel to manage the data it displays. By default, it uses a DefaultListModel, but you can create custom models by implementing the ListModel interface. This allows for dynamic and complex data handling.

JList generates ListSelectionEvent events when the selection changes. This allows developers to respond to user interactions, such as selecting or deselecting items in the list. JList is often used in conjunction with JScrollPane to handle large lists of data. The list automatically becomes scrollable when placed inside a JScrollPane.

The following are the commonly used constructors:

- **JList()**: Creates an empty JList.
- **JList(E[] listData)**: Creates a JList that displays the

elements of the specified array.

- **JList(Vector<? extends E> listData)**: Creates a JList that displays the elements in the specified Vector.
- **JList(ListModel<E> dataModel):** Creates a JList that displays elements from the specified ListModel.

The following are the commonly used methods:

Method	Description
void setListData(E[] listData)	Sets the contents of the list to the specified array of items.
void setListData(Vector<? extends E> listData)	Sets the contents of the list to the specified Vector of items.
void setModel(ListModel<E> dataModel)	Sets the data model for the list.
ListModel<E> getModel()	Returns the data model that provides the data for the list.
void setSelectionMode(int selectionMode)	Sets the selection mode for the list (SINGLE_SELECTION, SINGLE_INTERVAL_SELECTION, or MULTIPLE_INTERVAL_SELECTION).
int getSelectionMode()	Returns the current selection mode of the list.
void addListSelectionListener(ListSelectionListener l)	Adds a listener to receive selection events.
int[] getSelectedIndices()	Returns an array of indices of the selected items.
E getSelectedValue()	Returns the selected value in the list.
List<E> getSelectedValuesList()	Returns a list of selected values in the list.
void setSelectedIndex(int index)	Selects the item at the specified index.
void clearSelection()	Clears the current selection in the list.
void ensureIndexIsVisible(int index)	Ensures that the item at the specified index is visible, scrolling the list if necessary.

***Table 9.9**: Commonly used methods of JList*

The following example code demonstrates creating and using JList:

```
// Create a list of items
String[] items = {"Item 1", "Item 2", "Item 3", "Item 4", "Item 5"};
JList<String> list = new JList<>(items);
// Set selection mode
list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

// Add a ListSelectionListener to handle selection events
list.addListSelectionListener(new ListSelectionListener() {
    @Override
    public void valueChanged(ListSelectionEvent e) {
        if (!e.getValueIsAdjusting()) {
 System.out.println("Selected item: " +
list.getSelectedValue());
        }
```

```
    }
});
// Add the list to a scroll pane
JScrollPane scrollPane = new JScrollPane(list);
```

JComboBox

JComboBox is a Swing component that provides a combination of a drop-down list and a text field. It allows users to select an item from a list of options or to enter their own value if the text field is editable. This component is useful when you want to give users a choice of predefined options while still allowing them to input their own data. JComboBox class extends from **javax.swing.JComponent**. JComboBox can be set to editable or non-editable. When editable, users can type their own value into the text field. When non-editable, users can only select from the list of options. JComboBox uses a ComboBoxModel to manage its data. By default, it uses DefaultComboBoxModel, but you can provide a custom model to handle more complex scenarios. JComboBox generates action events when the user selects an item or changes the text (in editable mode). It also supports other event types, such as item events and focus events.

The following are the commonly used constructors:

- **JComboBox()**: Creates an empty combo box with no items.
- **JComboBox(E[] items)**: Creates a combo box with the specified array of items.
- **JComboBox(Vector<? extends E> items)**: Creates a combo box with the specified Vector of items.
- **JComboBox(ComboBoxModel<E> aModel)**: Creates a combo box with the specified ComboBoxModel.

The following are the commonly used methods:

Description	
void setModel(ComboBoxModel aModel)	mbo box.
ComboBoxModel getModel()	ovides the data for the combo box.
void addItem(E item)	mbo box.
void insertItemAt(E item, int index)	
void removeItem(Object item)	the combo box.
void removeItemAt(int index)	d index.
void setSelectedItem(Object anObject)	

Object getSelectedItem()	ted item.
void setEditable(boolean editable)	e.
boolean isEditable()	mbo box is editable, otherwise false.
void setRenderer(ListCellRenderer<? super E> renderer)	
ListCellRenderer<? super E> getRenderer()	
void setToolTipText(String text)	s when the user hovers over the combo box.
void addActionListener(ActionListener l)	nts, such as item selection.
void addItemListener(ItemListener l)	nts, such as selection changes.

***Table 9.11**: Commonly used methods of JCombobox*

The following code demonstrates creating and using JComboBox:

```
// Create an array of items
    String[] items = {"Option 1", "Option 2", "Option 3", "Option 4"};

// Create a JComboBox with the array of items
JComboBox<String> comboBox = new JComboBox<>(items);
comboBox.setBounds(50,50,100,200);

    // Set the JComboBox to editable
    comboBox.setEditable(true);

    // Add an ActionListener to handle selection changes

    // Add the combo box to the frame
    frame.add(comboBox);
```

Layout managers in containers

In Java Swing, Layout Managers are objects that control the size and position of components within a container (like JPanel or JFrame). They are essential for creating responsive, flexible, and well-organized user interfaces without needing to manually specify the absolute position and size of each component. In our programs in this chapter, we have not used any layout manager so far, so we need to specify the absolute position and size of each component by using **setBounds()** method. Layout managers are kept in **java.awt** package. A layout manager is responsible for positioning and sizing the components added to a container according to a specific layout

strategy. Java provides several built-in layout managers, each with its own behavior and use cases.

FlowLayout

FlowLayout is one of the simplest layout managers in Java Swing. It arranges the components in a container in a left-to-right flow, much like lines of text in a paragraph. If the container is not wide enough to fit all the components in one line, FlowLayout starts a new line. Components are placed in the order they were added to the container. You can specify alignment (left, center, right, leading, trailing). You can set horizontal and vertical gaps between components. Suitable for simple layouts where components are arranged in a row, like buttons in a toolbar. For setting FlowLayout for our container, the code syntax is as follows:

container.setLayout(new FlowLayout());

The following code snippet demonstrates how to use **FlowLayout**:

```
// Set the layout to FlowLayout
 frame.setLayout(new FlowLayout());
 // Add buttons to the frame
 frame.add(button1);
 frame.add(button2);
```

The layout manager is set for the JFrame using **frame.setLayout(new FlowLayout());**. Several buttons are created and added to the JFrame. The buttons are added to the frame one after the other, and FlowLayout arranges them horizontally. If the window is resized, the components will reflow automatically.

BorderLayout

BorderLayout divides the container into five regions: North, South, East, West, and Center. Each region can contain only one component, and the center region expands to fill any remaining space. The placement of regions is as follows:

- **North**: Top of the container.
- **South**: Bottom of the container.
- **East**: Right side of the container.
- **West**: Left side of the container.

- **Center**: Fills the remaining space.

This layout is Ideal for layouts where you need to position components relative to the edges of the container, such as in a typical application window. For setting BorderLayout for our container, the code syntax is as follows:

container.setLayout(new BorderLayout());

The following code snippet demonstrates the use of **BorderLayout**.

```
// Set the layout to BorderLayout
frame.setLayout(new BorderLayout());

// Add buttons to the frame in corresponding regions
frame.add(buttonNorth, BorderLayout.NORTH);
frame.add(buttonSouth, BorderLayout.SOUTH);
frame.add(buttonEast, BorderLayout.EAST);
frame.add(buttonWest, BorderLayout.WEST);
frame.add(buttonCenter, BorderLayout.CENTER);
```

The layout manager is set for the JFrame using **frame.setLayout(new BorderLayout());**.The buttons are added to the frame in the appropriate regions using **frame.add(component, region);**.

GridLayout

GridLayout arranges components in a grid of cells, with each cell having the same size. Components are placed in the grid row by row, left to right. You specify the number of rows and columns; the components are then arranged to fit the grid. You can set horizontal and vertical gaps between the cells. This layout manager is particularly useful when you want to arrange multiple components in a uniform, tabular format, such as a calculator interface or a form with evenly spaced fields. The following code snippet demonstrates **GridbagLayout**:

```
// Set GridLayout with 2 rows and 3 columns
frame.setLayout(new GridLayout(2, 3));
// Add buttons to the frame
frame.add(new JButton("Button 1"));
frame.add(new JButton("Button 2"));
```

```
frame.add(new JButton("Button 3"));
frame.add(new JButton("Button 4"));
frame.add(new JButton("Button 5"));
frame.add(new JButton("Button 6"));
```

The JFrame is configured to use a GridLayout with 2 rows and 3 columns, which divides the frame into a grid where each cell is evenly sized. Six JButton components, each labeled with a different number, are added to the frame. These buttons are automatically arranged in the grid layout, filling each cell sequentially from left to right and top to bottom.

CardLayout

CardLayout allows multiple components (or cards) to occupy the same space, with only one card visible at a time. It is like stacking components on top of each other and displaying one at a time. You can switch between the cards programmatically. Ideal for creating wizard-like interfaces or tabbed panes where only one panel should be visible at a time.

The following code snippet demonstrates using **CardLayout** in a panel.

```
// Create a JPanel with CardLayout
JPanel cardPanel = new JPanel(new CardLayout());
// Add panels to the cardPanel
cardPanel.add(panel1, "Panel1");
cardPanel.add(panel2, "Panel2");

// Add the cardPanel to the frame
frame.add(cardPanel);
```

A complete example demonstrating the use of the CardLayout manager to switch between two distinct panels within a single container is shared on the GitHub link.

GridBagLayout

GridBagLayout is a highly flexible and complex layout manager that arranges components in a grid of cells, but unlike GridLayout, the cells can have varying sizes, and components can span multiple rows or columns. Each component is associated with a

GridBagConstraints object that specifies its position, size, and how it should grow or shrink.

This layout is ideal for complex and non-uniform layouts where components need precise positioning and sizing, such as forms with labels, text fields, and buttons arranged irregularly.

The following code snippet demonstrates how to use **GridBagLayout** to arrange buttons in a grid with different row and column spans:

```
// Create a JPanel with GridBagLayout
JPanel panel = new JPanel(new GridBagLayout());
GridBagConstraints gbc = new GridBagConstraints();

// Define constraints for the first button
gbc.gridx = 0; // Column 0
gbc.gridy = 0; // Row 0
gbc.gridwidth = 2; // Span 2 columns
gbc.fill = GridBagConstraints.HORIZONTAL; // Fill the horizontal space
panel.add(new JButton("Button 1"), gbc);

// Define constraints for the second button
gbc.gridx = 2; // Column 2
gbc.gridy = 0; // Row 0
gbc.gridwidth = 1; // Span 1 column
gbc.fill = GridBagConstraints.HORIZONTAL; // Fill the horizontal space
panel.add(new JButton("Button 2"), gbc);

// Define constraints for the third button
gbc.gridx = 0; // Column 0
gbc.gridy = 1; // Row 1
gbc.gridwidth = 1; // Span 1 column
gbc.fill = GridBagConstraints.HORIZONTAL; // Fill the horizontal space
panel.add(new JButton("Button 3"), gbc);

// Define constraints for the fourth button
```

```
        gbc.gridx = 1; // Column 1
        gbc.gridy = 1; // Row 1
        gbc.gridwidth = 2; // Span 2 columns
        gbc.fill = GridBagConstraints.HORIZONTAL; // Fill the horizontal space
        panel.add(new JButton("Button 4"), gbc);
```

GridBagLayout allows precise control over component placement within a grid. Four JButton components are added to the panel, each positioned and sized according to GridBagConstraints. The constraints specify the button's grid location, spanning across columns, and how it should fill available space. For instance, the first button spans two columns, while the fourth button spans two columns in a different row. This layout manager adapts to the content and container size, making it ideal for complex interfaces requiring flexible alignment and spacing.

Event handling

Event handling in Java is the mechanism that allows your program to respond to various events such as user actions (like clicking a button, typing a key, or moving the mouse), system-generated events (like window resizing or closing), and other occurrences that affect the state of a program. By understanding and utilizing Java's event handling model, you can create responsive and interactive **graphical user interfaces (GUIs)** in your Java applications. Java provides a robust framework for handling events through its event-driven programming model. Java's event handling model is based on the delegation event model, where an event source generates an event and sends it to one or more listeners that process the event. Key components in event handling are:

- **Event**: An event is an object that describes a change in state in a source object. Events are generated by user interactions with GUI components (like JButton, JTextField, etc.), hardware devices (like keyboard, mouse), or system occurrences (like a timer going off).
- **Event source**: The source is the object that generates an event. Examples include GUI components like buttons, text fields, or windows. The source object registers listeners and

notifies them when an event occurs.

- **Event listener**: An event listener is an object that is notified when an event occurs. It must implement a specific interface depending on the type of event it wants to listen to. Listeners are registered with an event source, and when the event occurs, the appropriate method in the listener is invoked.
- **Event object**: Event objects contain information about the event, such as the event type and the source. These objects are passed to the listener methods when an event occurs.

  The event handling mechanism in Java follows a standard pattern:

  - **Event source**: The component that generates an event (e.g., a button).
  - **Event object**: The object that contains information about the event (e.g., ActionEvent).
  - **Event listener**: The object that listens for the event and defines how to respond (e.g., implementing ActionListener).

The process of handling events starts with Identifying the Event Source. It needs to be determined which component is generating the event. For example, if a button is clicked, the button is the event source. The next stage is to choose the appropriate event listener interface based on the type of event you want to handle (e.g., ActionListener for button clicks, KeyListener for keyboard events). Implement the required method(s) of the listener interface. Register the listener with the event source using a method like addActionListener, addKeyListener, etc. This establishes the connection between the event source and the listener. When the event occurs, the event source notifies the listener, and the listener's method is invoked. You can then write the code to handle the event.

Common event classes and listener interfaces

Java's event handling mechanism is managed through event classes and listener interfaces. They are provided by the **java.awt.event** package and are essential for creating responsive and well-behaved

GUI applications.

Common event classes represent different types of user interactions or system-generated events. For example, ActionEvent is triggered when a user performs an action like clicking a button, while MouseEvent handles mouse actions such as clicks, presses, or movements. To respond to these events, listener interfaces are implemented. These interfaces define methods that are invoked when the corresponding event occurs. For instance, ActionListener listens for ActionEvent and implements the actionPerformed method, while MouseListener listens for MouseEvent and provides methods like mouseClicked, mousePressed, and mouseReleased. By implementing these listener interfaces and registering them with the appropriate components, developers can create responsive and interactive applications that react to user inputs and other events.

Here are some of the most commonly used event classes and their corresponding listener interfaces.

Action event

Action events in Swing are triggered when a user interacts with GUI components such as buttons, menu items, or text fields. These events are handled using the ActionListener interface in Java's Swing framework.

- **Event class**: ActionEvent
- **Listener interface**: ActionListener
- **Common sources**: Buttons (JButton), menu items (JMenuItem), text fields (JTextField).
- **Method to implement**: void actionPerformed(ActionEvent e)

ActionEvent in Java is an event that occurs when a user interacts with a GUI component like a button, menu item, or text field, typically by clicking, pressing a key, or selecting an option. It is part of the Java Event Handling mechanism within AWT and Swing, enabling developers to respond to user actions. When an action is performed, an ActionEvent object is generated and passed to the actionPerformed method of the registered ActionListener, where the specific response to the action is defined. The event contains

information about the source of the action, the time it occurred, and an optional action command string, which can be used to identify the specific action performed, making it a crucial part of interactive Java applications.

Let us complete our GUI program with buttons to respond to the click event:

```
1. import javax.swing.JButton;
2. import javax.swing.JFrame;
3. import java.awt.event.ActionEvent;
4. import java.awt.event.ActionListener;
5. public class JButtonExample {
6.     public static void main(String[] args) {
7.         // Create a new JFrame container
8.         JFrame frame = new JFrame("JButton Example");
9.         frame.setSize(300, 200);
10.        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11.        frame.setLayout(null);
12.
13.        // Create a new JButton
14.        JButton button = new JButton("Click Me");
15.        button.setBounds(100, 50, 100, 30);
16.
17.        // Add an ActionListener to the button
18.        button.addActionListener(new ActionListener() {
19.            public void actionPerformed(ActionEvent e) {
20.                System.out.println("Button was clicked!");
21.            }
22.        });
23.
24.        // Add the button to the frame
25.        frame.add(button);
26.
27.        // Make the frame visible
28.        frame.setVisible(true);
29.    }
30. }
31.
```

Here is a sample Java program that changes the background color of a JFrame by selecting a JRadioButton:

```
1. import javax.swing.*;
2. import java.awt.*;
3. import java.awt.event.ActionEvent;
4. import java.awt.event.ActionListener;
5.
6. public class FrameColorChanger {
7.     public static void main(String[] args) {
8.         // Create a new JFrame
9.         JFrame frame = new JFrame("Frame Color Changer");
10.        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11.        frame.setSize(400, 300);
12.
13.        // Create radio buttons for different colors
14.        JRadioButton redButton = new JRadioButton("Red");
15.        JRadioButton greenButton = new JRadioButton("Green");
16.        JRadioButton blueButton = new JRadioButton("Blue");
17.
18.        // Group the radio buttons to ensure only one is selected at
    a time
19.        ButtonGroup group = new ButtonGroup();
20.        group.add(redButton);
21.        group.add(greenButton);
22.        group.add(blueButton);
23.
24.        // Create a panel for the radio buttons and add them to it
25.        JPanel radioPanel = new JPanel();
26.        radioPanel.add(redButton);
27.        radioPanel.add(greenButton);
28.        radioPanel.add(blueButton);
29.
30.        // Add ActionListeners to the radio buttons to change the
    color of the frame
31.        redButton.addActionListener(new ActionListener() {
32.            public void actionPerformed(ActionEvent e) {
33.                frame.getContentPane().setBackground(Color.RED);
34.            }
35.        });
36.
37.        greenButton.addActionListener(new ActionListener() {
38.            public void actionPerformed(ActionEvent e) {
```

```
frame.getContentPane().setBackground(Color.GREEN);
            }
        });

        blueButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                frame.getContentPane().setBackground(Color.BLUE);
            }
        });

        // Add the radio panel to the frame
        frame.add(radioPanel, BorderLayout.NORTH);

        // Set initial background color
        frame.getContentPane().setBackground(Color.WHITE);

        // Display the frame
        frame.setVisible(true);
    }
}
```

This program demonstrates how to change the background color of a JFrame using JRadioButton components in a Java Swing application. The JFrame is created with a size of *400x300* pixels and contains a panel at the top (**radioPanel**) where three radio buttons labeled "**Red**", "**Green**", and "**Blue**" are added. These buttons are grouped using a **ButtonGroup**, ensuring only one button can be selected at a time. Each radio button is linked to an **ActionListener**, which changes the frame's background color based on the selected option by calling **frame.getContentPane().setBackground(Color.RED)**, **Color.GREEN**, or **Color.BLUE**. The initial background color of the frame is set to white. When the user selects a radio button, the background color of the entire frame changes, accordingly, providing an interactive and visual way to modify the appearance of the GUI.

Let us see another Java program that uses a JCheckBox to change the alignment of a JLabel's text:

```java
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class LabelAlignmentChanger {
    public static void main(String[] args) {
        // Create a new JFrame
        JFrame frame = new JFrame("Label Alignment Changer");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 200);

        // Create a JLabel with some text
        JLabel label = new JLabel("Align me!", JLabel.CENTER);

        // Create a JCheckBox to toggle text alignment
        JCheckBox alignCheckBox = new JCheckBox("Align Left");

        // Add ActionListener to the checkbox to change label alignment
        alignCheckBox.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                if (alignCheckBox.isSelected()) {
                    label.setHorizontalAlignment(JLabel.LEFT);
                } else {
                    label.setHorizontalAlignment(JLabel.CENTER);
                }
            }
        });
       // Create a panel to hold the checkbox
        JPanel panel = new JPanel();
        panel.add(alignCheckBox);
              // Add the label and panel to the frame
        frame.add(label, BorderLayout.CENTER);
        frame.add(panel, BorderLayout.SOUTH);
        // Display the frame
        frame.setVisible(true);
    }
}
```

39.

This program illustrates how an actionEvent of JCheckBox can be used to change the alignment of text in a JLabel. The program sets up a JFrame containing a JLabel with the text "**Align me!**" and a checkbox labeled "**Align Left**". Initially, the label's text is centered within the frame. The JCheckBox is equipped with an ActionListener that listens for changes in the checkbox's state. When the checkbox is selected, the label's text alignment shifts to the left. When the checkbox is deselected, the text returns to its centered position. The label is placed at the center of the frame, and the checkbox is positioned at the bottom within a panel. This setup allows the user to interact with the checkbox and observe the immediate effect on the label's alignment, demonstrating how a simple GUI component, like a checkbox, can control and modify the properties of another component in a Swing application.

Mouse event

Mouse events occur when a user interacts with a GUI component using the mouse. These events are managed using the MouseListener and MouseMotionListener interfaces.

Event Class: MouseEvent

It provides information about the specific action performed by the mouse, including the type of event (click, press, release, etc.), the location of the cursor, and the state of the mouse buttons.

There are several types of mouse events that can be captured using the MouseEvent class. Each of these events corresponds to different actions performed with the mouse:

- Mouse Clicked (MOUSE_CLICKED)
- Mouse Pressed (MOUSE_PRESSED)
- Mouse Released (MOUSE_RELEASED)
- Mouse Entered (MOUSE_ENTERED)
- Mouse Exited (MOUSE_EXITED)
- Mouse Moved (MOUSE_MOVED)
- Mouse Dragged (MOUSE_DRAGGED)

Common Sources of mouse events are components that respond to

mouse actions, like JPanel, JLabel.

Key methods in MouseEvent

MouseEvent provides several methods to retrieve detailed information about the mouse event. Here are some of the most important methods:

- **getButton()**: Returns which mouse button was pressed (left, middle, or right button).
- **getClickCount()**: Returns the number of mouse clicks associated with this event (e.g., single-click, double-click).
- **getX() and getY()**: Return the X and Y coordinates of the mouse event relative to the source component.
- **getPoint()**: Returns a Point object representing the X and Y coordinates of the mouse event.
- **isPopupTrigger()**: Checks if the event should trigger a pop-up menu (often associated with a right-click on certain platforms).
- **getModifiersEx()**: Returns an extended set of modifiers indicating which additional keys (e.g., Shift, Ctrl) were pressed during the mouse event.

Key methods in MouseMotionEvent

The following are the key methods in MouseMotionEvent:

- **getX()**: Returns the X coordinate of the mouse cursor relative to the source component at the time of the event.
- **getY()**: Returns the Y coordinate of the mouse cursor relative to the source component at the time of the event.
- **getPoint():** Returns a Point object that contains the X and Y coordinates of the mouse cursor.
- **getModifiersEx()**: Returns an extended set of modifier keys (e.g., *Shift*, *Ctrl*) that were pressed during the event.

To handle mouse events, you need to implement the MouseListener and/or MouseMotionListener interfaces in your component or attach a MouseAdapter. These interfaces provide methods corresponding to the different types of mouse events:

- **MouseListener interface:**

- mouseClicked(MouseEvent e)
- mousePressed(MouseEvent e)
- mouseReleased(MouseEvent e)
- mouseEntered(MouseEvent e)
- mouseExited(MouseEvent e)

- **MouseMotionListener interface**:
  - mouseMoved(MouseEvent e)
  - mouseDragged(MouseEvent e)

You can attach these listeners to a component (like a JButton, JPanel, etc.) to handle the specific mouse events. Here is a simple example that demonstrates how to use MouseEvent to detect mouse clicks and display the coordinates:

```
1. import javax.swing.*;
2. import java.awt.event.*;
3. 
4. public class MouseEventExample extends JFrame {
5.     public MouseEventExample() {
6.         JLabel label = new JLabel("Click inside the window");
7.         label.setHorizontalAlignment(SwingConstants.CENTER);
8.         label.addMouseListener(new MouseListener() {
9.             public void mouseClicked(MouseEvent e) {
10.                 label.setText("Mouse clicked at (" + e.getX() + ", "
    + e.getY() + ")");
11.             }
12.             public void mousePressed(MouseEvent e) {}
13.             public void mouseReleased(MouseEvent e) {}
14.             public void mouseEntered(MouseEvent e) {}
15.             public void mouseExited(MouseEvent e) {}
16.         });
17.         add(label);
18.         setSize(300, 200);
19.         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20.         setVisible(true);
21.     }
22.     public static void main(String[] args) {
23.         new MouseEventExample();
24.     }
25. }
```

The **mouseClicked** method is implemented to update the label text with the X and Y coordinates of the mouse click. The other methods (**mousePressed**, **mouseReleased**, etc.) are left empty, as they are not needed for this example but must be included to satisfy the **MouseListener** interface. Here is a simple example demonstrating how to use **MouseMotionListener** to track mouse movement and dragging within a JPanel:

```
1. import javax.swing.*;
2. import java.awt.*;
3. import java.awt.event.*;
4. 
5. public class MouseMotionExample extends JFrame {
6.     private JLabel statusLabel;
7.     public MouseMotionExample() {
8.         // Create a label to display mouse coordinates
9.         statusLabel = new JLabel("Move the mouse or drag");
10. 
statusLabel.setHorizontalAlignment(SwingConstants.CENTER);
11.         // Create a panel to detect mouse events
12.         JPanel panel = new JPanel();
13.         panel.setPreferredSize(new Dimension(400, 300));
14.         panel.setBackground(Color.WHITE);
15.         // Add MouseMotionListener to the panel
16.         panel.addMouseMotionListener(new
MouseMotionListener() {
17.             @Override
18.             public void mouseMoved(MouseEvent e) {
19.                 statusLabel.setText("Mouse moved to (" + e.getX() +
", " + e.getY() + ")");
20.             }
21. 
22.             @Override
23.             public void mouseDragged(MouseEvent e) {
24.                 statusLabel.setText("Mouse dragged to (" + e.getX()
+ ", " + e.getY() + ")");
25.             }
26.         });
27.         // Add the panel and label to the frame
28.         add(panel, BorderLayout.CENTER);
29.         add(statusLabel, BorderLayout.SOUTH);
```

```
        // Set up the frame
        setTitle("MouseMotionEvent Example");
        setSize(500, 400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }
    public static void main(String[] args) {
        // Create and display the GUI
        new MouseMotionExample();
    }
}
```

This Java program sets up a JFrame with a JLabel for showing the current mouse coordinates and a JPanel that detects mouse motion events. By adding a **MouseMotionListener** to the panel, the program updates the label's text to reflect the cursor's position whenever the mouse moves or is dragged over the panel. The **mouseMoved** method updates the label with the current cursor coordinates when the mouse is moved without pressing any buttons, while the **mouseDragged** method updates it during dragging actions.

Key events

Key events in Java are used to handle keyboard interactions within a **graphical user interface** (**GUI**). They are crucial for applications that require user input via the keyboard, such as text editors, games, or any interactive application where keyboard shortcuts or commands are used. Different types of KeyEvents are:

- **KEY_PRESSED**: Triggered when a key is pressed down.
- **KEY_RELEASED**: Triggered when a key is released.
- **KEY_TYPED**: Triggered when a key is pressed and released and represents a character input. It is primarily used for character input in text fields and text areas.

KeyEvent provides several methods to obtain information about the key event:

- **getKeyCode()**: Returns the code of the key that was pressed or released. This code is an integer that represents the specific key (e.g., KeyEvent.VK_A for the 'A' key).

- **getKeyChar():** Returns the character corresponding to the key pressed, if applicable. This is useful for character input and is available during KEY_TYPED events.
- **getModifiersEx()**: Returns an extended set of modifier keys (e.g., Shift, Ctrl) that were pressed during the event.
- **getKeyLocation()**: Returns the location of the key on the keyboard, which can be useful for distinguishing between keys with multiple locations (e.g., left and right Shift keys).

To handle key events, you need to implement the KeyListener interface, which includes three methods:

- **keyPressed(KeyEvent e)**: Called when a key is pressed down. This method can be used to detect when a specific key or combination of keys is pressed.
- **keyReleased(KeyEvent e)**: Called when a key is released. This method is often used to detect the completion of a key action or to handle the release of a key.
- **keyTyped(KeyEvent e)**: Called when a key is pressed and released, generating a character. This method is typically used for character input in text components.

Window events

Window events are used to manage interactions with top-level window components, such as JFrame, JDialog, and JWindow. These events are crucial for handling window state changes, such as opening, closing, minimizing, and activating. Key classes and interfaces for window events are discussed below:

- **WindowEvent class**: WindowEvent is the primary class used to represent events related to changes in the state of a window. It extends ComponentEvent and provides information about window-specific actions.
- **WindowListener interface**: The WindowListener interface is used to handle various window events. It includes the following methods, which correspond to different types of window interactions:
    - windowOpened(WindowEvent e):

- windowClosing(WindowEvent e):
- windowClosed(WindowEvent e):
- windowIconified(WindowEvent e):
- windowDeiconified(WindowEvent e):
- windowActivated(WindowEvent e):
- windowDeactivated(WindowEvent e):

Event adapters

For some listener interfaces that have multiple methods (like **MouseListener** or **WindowListener**), Java provides adapter classes (e.g., **MouseAdapter**, **WindowAdapter**). These are abstract classes that provide empty implementations of all the methods in the listener interface, allowing you to override only the methods you are interested in. They are part of the **java.awt.event** and **javax.swing.event** packages and serve as a bridge between event sources and event handlers. Some common adapter classes are:

- **WindowAdapter**: Provides default implementations of methods in the WindowListener interface. Useful for handling window events such as opening, closing, minimizing, and activating.
- **MouseAdapter**: Provides default implementations of methods in the MouseListener and MouseMotionListener interfaces. Useful for handling mouse events such as clicks, presses, releases, movements, and drags.
- **KeyAdapter**: Provides default implementations of methods in the KeyListener interface. Useful for handling keyboard events such as key presses, releases, and typing.
- **FocusAdapter**: Provides default implementations of methods in the FocusListener interface. Useful for handling focus events when a component gains or loses focus.
- **ComponentAdapter**: Provides default implementations of methods in the ComponentListener interface. Useful for handling component events such as resizing, moving, or showing.

Here is a basic example demonstrating how to handle the window

closing event using both WindowListener and WindowAdapter. The example will display a confirmation dialog when the user attempts to close the window, asking if they want to save any unsaved changes.

```
1. import javax.swing.*;
2. import java.awt.event.*;
3. public class WindowListenerExample extends JFrame {
4.     private boolean unsavedChanges = true; // Simulate unsaved changes
5.     public WindowListenerExample() {
6.         setTitle("WindowListener Example");
7.         setSize(400, 300);
8.         setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
9.         // Add WindowListener to handle window events
10.        addWindowListener(new WindowListener() {
11.            @Override
12.            public void windowOpened(WindowEvent e) {
13.                // Do nothing
14.            }
15.            @Override
16.            public void windowClosing(WindowEvent e) {
17.                if (unsavedChanges) {
18.                    int choice = JOptionPane.showConfirmDialog(
19.                        WindowListenerExample.this,
20.                        "You have unsaved changes. Do you want to save before closing?",
21.                        "Confirm Exit",
22.                        JOptionPane.YES_NO_CANCEL_OPTION
23.                    );
24.                    if (choice == JOptionPane.YES_OPTION) {
25.                        // Simulate saving changes
26.                        System.out.println("Saving changes...");
27.                        System.exit(0);
28.                    } else if (choice == JOptionPane.NO_OPTION) {
29.                        System.exit(0);
30.                    }
31.                    // If CANCEL_OPTION, do nothing
32.                } else {
33.                    System.exit(0);
```

```
            }
        }
        @Override
        public void windowClosed(WindowEvent e) {
            // Do nothing
        }
        @Override
        public void windowIconified(WindowEvent e) {
            // Do nothing
        }
        @Override
        public void windowDeiconified(WindowEvent e) {
            // Do nothing
        }
        @Override
        public void windowActivated(WindowEvent e) {
            // Do nothing
        }
        @Override
        public void windowDeactivated(WindowEvent e) {
            // Do nothing
        }
    });
    setVisible(true);
  }
  public static void main(String[] args) {
    new WindowListenerExample();
  }
}
```

Let us rewrite the program using the corresponding adapter class:

```
import javax.swing.*;
import java.awt.event.*;
public class WindowAdapterExample extends JFrame {
  private boolean unsavedChanges = true; // Simulate unsaved changes
  public WindowAdapterExample() {
    setTitle("WindowAdapter Example");
    setSize(400, 300);
    setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
```

```
        // Add WindowListener using WindowAdapter
        addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                if (unsavedChanges) {
                    int choice = JOptionPane.showConfirmDialog(
                        WindowAdapterExample.this,
                        "You have unsaved changes. Do you want to save before closing?",
                        "Confirm Exit",
                        JOptionPane.YES_NO_CANCEL_OPTION
                    );
                    if (choice == JOptionPane.YES_OPTION) {
                        // Simulate saving changes
                        System.out.println("Saving changes...");
                        System.exit(0);
                    } else if (choice == JOptionPane.NO_OPTION) {
                        System.exit(0);
                    }
                    // If CANCEL_OPTION, do nothing
                } else {
                    System.exit(0);
                }
            }
        });
        setVisible(true);
    }
    public static void main(String[] args) {
        new WindowAdapterExample();
    }
}
```

Using **WindowListener** and **WindowAdapter** for handling window events in Java each has distinct advantages and trade-offs. **WindowListener** requires implementing all methods of the interface, which can lead to more boilerplate code and reduced readability when only a few methods are needed. This approach can be cumbersome if you are interested in handling only specific events, as it necessitates providing empty implementations for unused methods. On the other hand, **WindowAdapter** simplifies

the process by offering default implementations for all methods, allowing you to override only those that are relevant to your application. This results in cleaner, more focused code and easier maintenance. While **WindowListener** provides a more comprehensive approach suitable for handling a broad range of events, **WindowAdapter** is preferred for its conciseness and ease of use, particularly when only a subset of events needs to be managed.

Conclusion

This chapter has provided a comprehensive overview of Java Swing, focusing on the essential components, containers, layout managers, and event handling mechanisms that form the foundation of GUI development. By understanding the hierarchy and roles of containers, both top-level and intermediate, you have learned how to structure your applications effectively. The detailed exploration of components like JLabel, JTextField, JList, and JComboBox has equipped you with the tools to create interactive and user-friendly interfaces. The various layout managers, from FlowLayout to GroupLayout, allow you to design flexible and organized layouts tailored to your application's needs. Finally, mastering event handling, including event sources, listeners, and adapter classes, empowers you to create responsive and dynamic applications that react seamlessly to user interactions.

In the next chapter, we will learn database connectivity with JDBC.

Exercises

1. Create a basic Swing application with a JFrame containing a JLabel that displays "Hello, World!". Add a JButton to the JFrame. When the button is clicked, the text of the JLabel should change to "Button Clicked!".
2. Create a Swing application with a JTextField and a JLabel. When the user types text into the JTextField and presses Enter, the JLabel should display the entered text.
3. Create a GUI with three JButtons labeled "Red", "Green", and "Blue". When each button is clicked, change the background color of the JFrame to the corresponding color.

4. Develop a Swing application with a JComboBox containing a list of programming languages. Below the combo box, place a JTextArea. When the user selects a language from the combo box, display a brief description of the selected language in the text area.
5. Design a simple calculator using JButtons for digits (0-9) and basic operators (+, -, *, /). Use a JTextField to display the input and result. Implement the functionality to perform basic arithmetic operations.
6. Create a to-do list application where users can add tasks using a JTextField and a JButton. Display the tasks in a JList and add functionality to remove tasks from the list when they are completed.
7. Create an application that uses JTabbedPane to organize content. Each tab should display different content.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 10
Database Connectivity with JDBC

Introduction

In today's data-driven world, managing and accessing vast amounts of information efficiently is crucial for any application. Databases play a vital role in storing, retrieving, and managing data, making them a central component of most software systems. To interact with these databases programmatically, developers need a reliable and standard approach that allows their applications to communicate seamlessly with various **database management systems (DBMS)**. This is where **Java Database Connectivity (JDBC)** comes into play.

JDBC is an essential part of the Java Standard Edition and provides a comprehensive API for connecting Java applications to a wide range of databases. Whether you are working with MySQL, Oracle, PostgreSQL, or any other relational database, JDBC offers a uniform interface to execute SQL queries, update records, and retrieve results.

This chapter will guide you through the fundamental concepts of

JDBC, starting with an overview of its architecture and core components. You will learn how to establish a connection with a database, execute SQL statements, and handle the results in your Java applications. JDBC is a specification that provides a comprehensive set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executables, such as Applications, Applets, Servlets, **Java Server Pages (JSPs), Enterprise JavaBeans (EJBs)**. All of these different executables can use a JDBC driver to access a database and take advantage of the stored data. Additionally, we will explore advanced topics such as handling transactions, using prepared statements to prevent SQL injection, and connecting to different types of databases. By the end of this chapter, you will have a solid understanding of how to integrate databases into your Java applications using JDBC, enabling you to build more dynamic and data-driven software solutions.

Structure

The chapter covers the following topics:

- Java Database Connectivity architecture
- Types of JDBC drivers
- Major JDBC classes and interface
- Database operations using JDBC
- Creating a simple JDBC application
- Using PreparedStatement
- Using CallableStatement
- Exploring ResultSet operations
- JDBC with Swing
- Mapping between Java data types and SQL data types
- Using advanced data types

Objectives

This chapter aims to provide a comprehensive understanding of JDBC by covering its architecture and the various types of JDBC

drivers available for database interaction. You will explore the major JDBC classes and interfaces, gaining practical experience with fundamental database operations such as inserting, updating, and querying data. The chapter will guide you through creating a simple JDBC application, utilizing PreparedStatement for parameterized queries, and CallableStatement for executing stored procedures. Additionally, you will learn how to integrate JDBC with Swing GUI to create user-friendly applications. The chapter will also cover the mapping between Java data types and SQL data types and introduce advanced data types like BLOB and CLOB, enabling you with the skills to handle complex data interactions in Java applications.

Java Database Connectivity architecture

JDBC is a Java-based API that allows developers to interact with databases from their Java applications. It provides a standardized way to connect to databases, execute SQL queries, and retrieve results. Understanding the architecture of JDBC is essential for effectively utilizing this API in your applications. The JDBC library combines APIs for various database related tasks such as creating a connection to a database, creating SQL or MySQL statements, executing SQL or MySQL queries in the database, and viewing and modifying the resulting records. JDBC is designed with a layered architecture that consists of two main layers:

- **JDBC application layer**: This is the top layer where the Java application interacts with the JDBC API. It includes the classes and interfaces provided by the JDBC API, such as DriverManager, Connection, Statement, PreparedStatement, ResultSet, and others. These classes and interfaces are used by the application to connect to the database, execute SQL commands, and process the results.
- **JDBC driver layer**: This is the lower layer that acts as a bridge between the JDBC API and the database. The JDBC Driver translates the high-level JDBC API calls into database-specific calls, allowing the application to communicate with various databases without needing to know the details of the underlying database. JDBC drivers are provided by database vendors or third-party developers.

JDBC architecture is shown diagrammatically as follows:

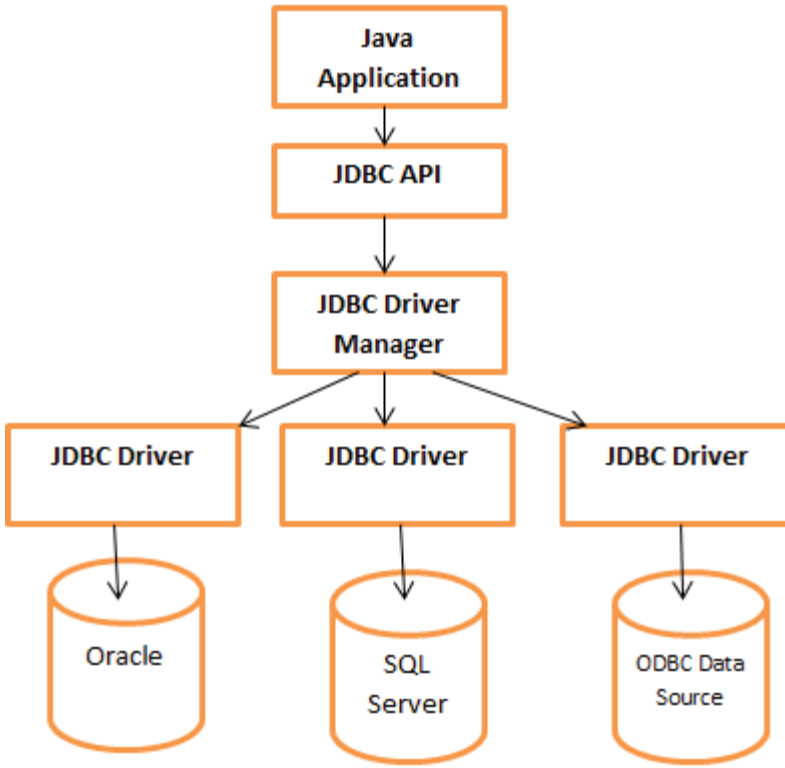


***Figure 10.1**: JDBC architecture*

Advantages of JDBC architecture

The following are the advantages of JDBC architecture:

- **Database independence**: JDBC provides a uniform interface to connect to different databases, allowing developers to write database-agnostic code.
- **Standardization**: JDBC is part of the Java Standard Edition, ensuring consistency and compatibility across Java applications.
- **Portability**: JDBC drivers are platform-independent (except *Type 1* and *Type 2*), allowing Java applications to run on different platforms without modification.
- **Flexibility**: JDBC supports various types of databases,

including relational, NoSQL, and even in-memory databases, through different types of drivers.

Understanding the JDBC architecture is crucial for building robust and scalable database-driven applications. By leveraging JDBC, developers can efficiently manage database connections, execute queries, and handle transactions, ensuring seamless data access and manipulation in their Java applications.

Types of JDBC drivers

JDBC drivers are essential components that allow Java applications to communicate with various databases. These drivers translate the JDBC API calls into database-specific operations. There are four main types of JDBC drivers, each with its unique architecture, advantages, and limitations. Let us explore these types in detail.

JDBC-ODBC bridge driver

JDBC-ODBC bridge driver is also called a type I driver. It acts as an intermediary between Java applications and ODBC-compliant databases. It allows Java programs to connect to databases using the ODBC interface.

- **Architecture**:
  - The Type 1 driver uses a combination of JDBC and **Open Database Connectivity (ODBC)** technologies to connect to databases. When a Java application makes a JDBC call, the Type 1 driver translates it into an ODBC function call, which is then processed by the ODBC driver.
  - The ODBC driver communicates with the database using the ODBC API.
- **Advantages**:
  - **Quick setup**: Since ODBC drivers are widely available and supported by many databases, setting up a Type 1 driver is relatively straightforward.
  - **Good for learning**: Type 1 drivers are useful for learning and prototyping JDBC as they require

minimal setup.

- **Limitations**:
  - **Platform dependence**: The Type 1 driver relies on the ODBC driver, which is platform dependent. This limits the portability of the application.
  - **Performance overhead**: The double translation (JDBC to ODBC to database) introduces performance overhead, making Type 1 drivers slower than other types.
  - **Outdated**: Type 1 drivers are considered outdated and are rarely used in modern applications due to the availability of more efficient driver types.
- **Use cases**: Type 1 drivers are typically used in legacy systems or for applications that need to connect to databases that only have ODBC drivers available.

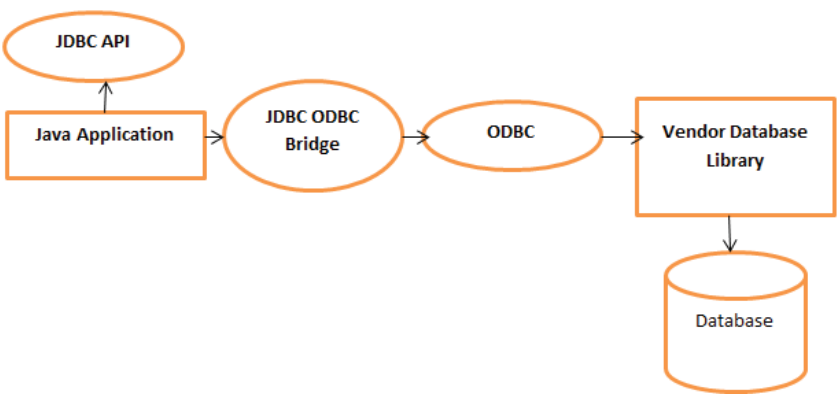


***Figure 10.2**: Type I JDBC driver*

Native-API Driver

The Native-API Driver (Partially Java Driver) is a Type 2 JDBC driver that bridges Java applications and databases using native APIs provided by database vendors. It translates JDBC calls into database-specific calls using a native client library.

- **Architecture**:
  - The Type 2 driver uses native database client libraries

to communicate with the database.

- JDBC calls are translated into native API calls specific to the database (e.g., **Oracle Call Interface** (**OCI**) for Oracle databases).
- These native calls are then processed by the database client libraries to interact with the database.

- **Advantages:**
  - **Better performance**: Since the Type 2 driver uses native database libraries, it offers better performance compared to the Type 1 driver.
  - **Access to database-specific features**: Type 2 drivers can take advantage of database-specific features and optimizations provided by the native libraries.
- **Limitations:**
  - **Platform dependence**: The reliance on native libraries makes the Type 2 driver platform-dependent, limiting the portability of the application.
  - **Complex setup**: Setting up Type 2 drivers requires installing and configuring native database client libraries on the client machine, which can be complex.
- **Use cases**: Type 2 drivers are typically used in environments where performance is critical and where database-specific features are required.

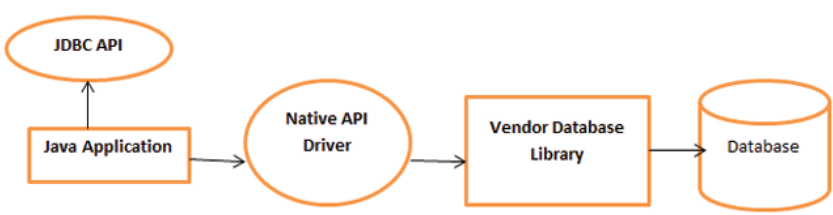


***Figure 10.3**: Type II JDBC driver Native-API Driver (Partially Java Driver)*

Type 3: Network Protocol Driver

The Network Protocol Driver (Type 3 JDBC Driver) is a fully Java-based driver that connects Java applications to databases using a

middleware server. The driver translates JDBC calls into a database-independent network protocol, which the middleware server further converts into database-specific calls.

- **Architecture**:
  - The Type 3 driver introduces a middle-tier server that acts as an intermediary between the Java application and the database.
  - JDBC calls from the application are sent to the middle-tier server using a network protocol (e.g., HTTP, RMI).
  - The middle-tier server translates the JDBC calls into the database-specific protocol and communicates with the database.
- **Advantages**:
  - **Platform independence**: Since the driver is written entirely in Java and uses network communication, it is platform independent.
  - **Centralized management**: The middle-tier server can manage database connections and load balancing, making it easier to maintain and scale the application.
  - **Better performance over networks**: Type 3 drivers are well-suited for multi-tiered applications, particularly when the client and database are on different networks.
- **Limitations**:
  - **Network overhead**: The use of a middle-tier server introduces additional network overhead, which can affect performance, especially in low-bandwidth environments.
  - **Complex architecture**: The architecture is more complex due to the involvement of a middle-tier server, requiring additional setup and maintenance.
- **Use cases**: Type three drivers are typically used in enterprise-level applications that require distributed architecture, centralized management, and scalability.

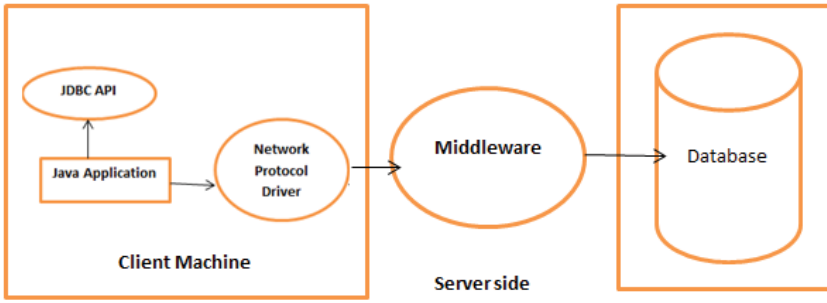


***Figure 10.4**: Type III JDBC driver-Network Protocol Driver*

Thin Driver

The Thin Driver (**Pure Java Driver**) is a Type 4 JDBC driver that directly communicates with the database using the database's native protocol. It is entirely written in Java, eliminating the need for any native libraries or middleware components.

- **Architecture**:
  - The Type 4 driver, also known as the thin driver, is written entirely in Java and converts JDBC calls directly into the database-specific protocol.
  - This driver communicates directly with the database without the need for any native libraries or middleware.
- **Advantages**:
  - **Platform independence**: Being a pure Java driver, the Type 4 driver is platform-independent and can run on any system with a JVM.
  - **High performance**: The direct communication with the database and the absence of additional translation layers result in high performance.
  - **Ease of use**: The Type 4 driver is easy to set up and deploy since it does not require any native libraries or middleware.
- **Limitations**:
  - **Database-specific:** Since the Type 4 driver

communicates directly with the database, each driver is specific to a particular database. A separate Type 4 driver is needed for each database.

- **Use cases**: Type 4 drivers are the most commonly used drivers in modern applications due to their high performance, ease of use, and portability. They are ideal for both standalone and enterprise-level applications.

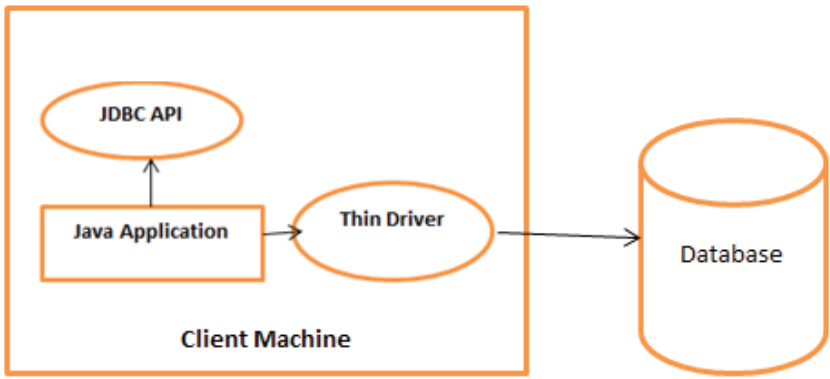


***Figure 10.5**: Type 4: Thin driver (Pure Java driver)*

Each type of JDBC driver has its own strengths and weaknesses, making them suitable for different scenarios:

- **Type 1**: Suitable for learning or prototyping with legacy systems but not recommended for production use due to performance and platform dependency.
- **Type 2**: Offers better performance and access to database-specific features but is limited by platform dependency and complex setup.
- **Type 3**: Ideal for distributed, multi-tiered applications with centralized management but introduces network overhead and complexity.
- **Type 4**: The most widely used and recommended driver for modern applications, providing high performance, ease of use, and platform independence.

Choosing the right JDBC driver depends on the specific needs of your application, including performance requirements, platform

considerations, and the database management system in use.

Major JDBC classes and interface

JDBC provides a set of classes and interfaces that allow Java applications to interact with relational databases. These classes and interfaces form the core of the JDBC API, enabling developers to connect to databases, execute SQL queries, and process the results. Understanding these components is crucial for effective database programming in Java. The JDBC API provides a rich set of classes and interfaces to interact with relational databases. The JDBC API is divided into two main packages: **java.sql**, which provides the core functionality for database operations, and **javax.sql**, which extends these capabilities for more advanced, enterprise-level applications. The **java.sql** package is the core package of the JDBC API, containing the primary classes and interfaces needed for database operations. This package allows Java applications to connect to a database, execute SQL statements, and process the results.

DriverManager

The DriverManager class is responsible for managing a list of database drivers. It acts as a central mechanism to establish connections between a Java application and a database. Typically used to establish a connection to a database, which is the first step in any JDBC application. Some of its commonly used methods are as follows:

Method	Description
static Connection getConnection(String url)	Attempts to establish a connection to the database using the specified URL.
static Connection getConnection(String url, String user, String password)	Attempts to establish a connection to the specified URL, username, and password.
static Connection getConnection(String url, Properties info)	Attempts to establish a connection to the database using the specified URL and a set of connection properties.
static void registerDriver(Driver driver)	Registers the given driver with the DriverManager. This method can be used to manually load a JDBC driver.
static Enumeration<Driver> getDrivers()	Retrieves an enumeration of all the registered JDBC drivers.
static void deregisterDriver(Driver driver)	Removes the specified driver from the DriverManager's list of registered drivers.
static void setLoginTimeout(int seconds)	Sets the maximum time in seconds that a driver will wait while attempting to establish a connection.
static int getLoginTimeout()	Retrieves the maximum time in seconds that a driver will wait while attempting to establish a connection.

Method	Description
`static void setLogWriter(PrintWriter out)`	Sets the log writer for DriverManager, which can be used for logging errors and information messages.
`static PrintWriter getLogWriter()`	Retrieves the log writer for DriverManager, used for logging errors and information messages.
`static void setLogStream(PrintStream out)`	Sets the log stream for DriverManager for logging messages. Prefer setLogWriter() instead.
`static PrintStream getLogStream()`	Retrieves the log stream for DriverManager. Prefer getLogWriter() instead.
`static Logger getParentLogger()`	Returns the parent logger for the loggers used by the JDBC drivers.
`static void println(String message)`	Prints a message to the log writer or log stream (for debugging purposes).

***Table 10.1**: Commonly used methods of DriverManager class*

Connection

The Connection interface represents an active connection to a database. It provides methods to create SQL statements, manage transactions, and control connection properties. The Connection object is central to any JDBC operation, as it is used to send commands to the database and manage transactions. Some of its commonly used methods are listed below:

Method	Description
`void close()`	Closes the database connection, releasing all associated resources.
`void commit()`	Commits the current transaction, making all changes since the last commit or rollback permanent.
`Statement createStatement()`	Creates a Statement object for executing simple SQL queries.
`PreparedStatement prepareStatement(String sql)`	Creates a PreparedStatement object for executing parameterized SQL queries.
`CallableStatement prepareCall(String sql)`	Creates a CallableStatement object for executing stored procedures.
`boolean getAutoCommit()`	Checks the current auto-commit mode.
`void setAutoCommit(boolean autoCommit)`	Sets the auto-commit mode. If true, each SQL statement is committed automatically.
`boolean isClosed()`	Checks if the connection is closed.
`DatabaseMetaData getMetaData()`	Retrieves metadata about the database, such as tables, columns, and supported SQL features.
`void rollback()`	Rolls back the current transaction, undoing all changes since the last commit or rollback.
`void setTransactionIsolation(int level)`	Sets the transaction isolation level for the connection.
`void setReadOnly(boolean readOnly)`	Configures the connection to be read-only, which can optimize performance for read-only queries.
`boolean isReadOnly()`	Checks if the connection is in read-only mode.
`Savepoint setSavepoint()`	Creates a savepoint in the current transaction.
`Savepoint setSavepoint(String name)`	Creates a named savepoint in the current transaction.
`void rollback(Savepoint savepoint)`	Rolls back to the specified savepoint.

Method	Description
`void releaseSavepoint(Savepoint savepoint)`	Removes the specified savepoint from the current transaction.
`Clob createClob()`	Constructs a Clob (Character Large Object) for storing large character data.
`Blob createBlob()`	Constructs a Blob (Binary Large Object) for storing large binary data.
`NClob createNClob()`	Constructs an NClob (National Character Large Object) for storing large character data in a national character set.
`boolean isValid(int timeout)`	Checks if the connection is still valid, with a specified timeout period.
`void abort(Executor executor)`	Terminates the connection and releases its resources immediately.

Table 10.2: Commonly used methods of Connection interface

Statement

The Statement interface is used to execute SQL queries against the database. It is used for executing simple SQL queries without parameters. Statement objects are used for executing SQL queries that do not require input parameters. Some of its commonly used methods are listed below:

Method	Description
`ResultSet executeQuery(String sql)`	Executes the given SQL query and returns a ResultSet object that contains the data produced by the query.
`int executeUpdate(String sql)`	Executes the given SQL statement (like an INSERT, UPDATE, or DELETE), and returns the number of affected rows.
`boolean execute(String sql)`	Executes the given SQL statement, which may return multiple results, and returns true if the first result is a ResultSet; false if it is an update count or there are no results.
`void close()`	Closes the Statement object, releasing its resources.
`void setMaxRows(int max)`	Sets the maximum number of rows that a ResultSet object can contain. Additional rows are silently discarded.
`int getMaxRows()`	Retrieves the maximum number of rows that a ResultSet object can contain.
`void setQueryTimeout(int seconds)`	Sets the number of seconds the driver will wait for a Statement to execute before throwing a SQLException.
`int getQueryTimeout()`	Retrieves the number of seconds the driver will wait for a Statement to execute before throwing a SQLException.
`void addBatch(String sql)`	Adds the given SQL command to the current list of commands for this Statement object. This method is used to batch multiple SQL commands together.
`int[] executeBatch()`	Executes all the commands in the current batch of commands and returns an array of update counts, one for each command.
`void clearBatch()`	Clears the current list of SQL commands for this Statement object.
`Connection getConnection()`	Retrieves the Connection object that produced this Statement object.
`void setFetchSize(int rows)`	Gives the JDBC driver a hint as to the number of rows that should be fetched from the database when more rows are needed.
`int getFetchSize()`	Retrieves the fetch size for this Statement object.
`void setFetchDirection(int direction)`	Gives the JDBC driver a hint as to the direction in which rows should be processed.

Method	Description
int getFetchDirection()	Gets the fetch direction for this Statement object.
boolean getMoreResults()	Moves to a Statement object that might return multiple results, and returns true if the next result is a ResultSet object; false if it is an update count or there are no more results.
int getUpdateCount()	Retrieves the result as an update count; returns -1 if the current result is a ResultSet object or there are no more results.

***Table 10.3**: Commonly used methods of Statement interface*

These methods cover the essential operations for executing SQL statements, managing results, and handling batched commands using a **Statement** object. For example, a query can be executed using the statement object as follows:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM employees");
```

PreparedStatement

The PreparedStatement interface extends Statement and is used for executing parameterized SQL queries. It allows SQL queries to be precompiled, which improves performance and security. PreparedStatement is preferred over Statement for executing queries that require input parameters, as it helps prevent SQL injection attacks and improves performance. Some commonly used methods of PreparedStatement are as follows:

Method	Description
void setInt(int parameterIndex, int value)	Sets the value of the designated parameter using the given int value.
void setString(int parameterIndex, String value)	Sets the value of the designated parameter using the given String value.
void setDouble(int parameterIndex, double value)	Sets the value of the designated parameter using the given double value.
void setDate(int parameterIndex, Date value)	Sets the value of the designated parameter using the given Date value.
void setBoolean(int parameterIndex, boolean value)	Sets the value of the designated parameter using the given Boolean value.
void setLong(int parameterIndex, long value)	Sets the value of the designated parameter using the given long value.
void setFloat(int parameterIndex, float value)	Sets the value of the designated parameter using the given float value.
void setBytes(int parameterIndex, byte[] value)	Sets the value of the designated parameter using the given byte array.
void setObject(int parameterIndex, Object value)	Sets the value of the designated parameter using the given Object. This method is useful for setting parameters dynamically.
ResultSet executeQuery()	Executes the SQL query in this PreparedStatement object and returns a ResultSet object that contains the result.
int executeUpdate()	Executes the SQL statement in this PreparedStatement object, which must be an SQL INSERT, UPDATE, or DELETE statement; returns the number of affected rows.
boolean execute()	Executes the SQL statement in this PreparedStatement object, which may return multiple results; returns true if the first result is a ResultSet object, false if it is an update count or there are no results.
void clearParameters()	Clears the parameter values immediately.

Method	Description
void setNull(int parameterIndex, int sqlType)	Sets the parameter at the specified index to SQL NULL.
void addBatch()	Adds a set of parameters to the batch of commands for this PreparedStatement object.
void setTimestamp(int parameterIndex, Timestamp value)	Sets the parameter at the specified index to the given Timestamp value.
void setTime(int parameterIndex, Time value)	Sets the parameter at the specified index to the given Time value.
void setBigDecimal(int parameterIndex, BigDecimal value)	Sets the parameter at the specified index to the given BigDecimal value.
void setBlob(int parameterIndex, Blob value)	Sets the parameter at the specified index to the given Blob object.
void setClob(int parameterIndex, Clob value)	Sets the parameter at the specified index to the given Clob object.

***Table 10.4**: Commonly used methods of PreparedStatement interface*

The below listed code snippet shows an example use of **PreparedStatement** object:

```
PreparedStatement pstmt = conn.prepareStatement("INSERT INTO employees (name, age, department) VALUES (?, ?, ?)");
pstmt.setString(1, "John Doe");
pstmt.setInt(2, 30);
pstmt.setString(3, "HR");
pstmt.executeUpdate();
```

CallableStatement

The CallableStatement interface is used to execute stored procedures in the database. It allows the execution of database procedures and functions that are stored on the database server. CallableStatement is used when working with stored procedures that encapsulate complex logic within the database. Here are some commonly used methods of CallableStatement interface:

Method	Description
void registerOutParameter(int parameterIndex, int sqlType)	Registers the parameter at the specified index as an output parameter of the given SQL type.
void registerOutParameter(int parameterIndex, int sqlType, int scale)	Registers the parameter at the specified index as an output parameter of the given SQL type with a specified scale.
boolean wasNull()	Returns whether the last column read had a value of SQL NULL. Returns true if the last column read was SQL NULL; otherwise false.
void setInt(int parameterIndex, int x)	Sets the parameter at the specified index to the given int value.
void setString(int parameterIndex, String x)	Sets the parameter at the specified index to the given String value.
void setDouble(int parameterIndex, double x)	Sets the parameter at the specified index to the given double value.
void setDate(int parameterIndex, Date x)	Sets the parameter at the specified index to the given Date value.
void setBoolean(int parameterIndex, boolean x)	Sets the parameter at the specified index to the given Boolean value.

void setLong(int parameterIndex, long x)	Sets the specified index to the given long value.
void setFloat(int parameterIndex, float x)	Sets the specified index to the given float value.
void setBytes(int parameterIndex, byte[] x)	Sets the specified index to the given byte array.
void setObject(int parameterIndex, Object x)	Sets the specified index to the given Object.
ResultSet executeQuery()	Executes the SQL query in this CallableStatement object and returns a ResultSet object containing the result.
int executeUpdate()	Executes the SQL statement in this CallableStatement object, which must be an SQL INSERT, UPDATE, or DELETE statement; returns the number of affected rows.
boolean execute()	Executes the SQL statement in this CallableStatement object, which may return multiple results; returns true if the first result is a ResultSet object, false if it is an update count or there are no results.
void setNull(int parameterIndex, int sqlType)	Sets the specified index to SQL NULL.
void setTimestamp(int parameterIndex, Timestamp x)	Sets the specified index to the given Timestamp value.
void setTime(int parameterIndex, Time x)	Sets the specified index to the given Time value.
void setBigDecimal(int parameterIndex, BigDecimal x)	Sets the specified index to the given BigDecimal value.
void setBlob(int parameterIndex, Blob x)	Sets the specified index to the given Blob object.
void setClob(int parameterIndex, Clob x)	Sets the specified index to the given Clob object.

***Table 10.5**: Commonly used methods of CallableStatement interface*

The following code block shows an example usage of **CallableStatement**:

```
CallableStatement cstmt = conn.prepareCall("{call
getEmployeeCount(?)}");
cstmt.registerOutParameter(1, java.sql.Types.INTEGER);
cstmt.execute();
int count = cstmt.getInt(1);
```

ResultSet

The ResultSet interface represents the result set of a SQL query. It provides methods to navigate through the results and retrieve data from the columns. ResultSet is used to iterate over and process the rows returned by a SQL query. Here are the key methods in the ResultSet interface, along with their descriptions:

Method	Description
boolean next()	Moves the cursor to the next row in the ResultSet and returns true if there is a row, false if there are no more rows.

boolean previous()	Moves the cursor to the previous row in the ResultSet. Returns true if the cursor is on a valid row; false if it is off the result set.
boolean first()	Moves the cursor to the first row in the ResultSet. Returns true if the cursor is on a valid row; false if the ResultSet is empty.
boolean last()	Moves the cursor to the last row in the ResultSet. Returns true if the cursor is on a valid row; false if the ResultSet is empty.
int getInt(int columnIndex)	Retrieves the value of the specified column as an int.
int getInt(String columnLabel)	Retrieves the value of the specified column (by name) as an int.
String getString(int columnIndex)	Retrieves the value of the specified column as a String.
String getString(String columnLabel)	Retrieves the value of the specified column (by name) as a String.
double getDouble(int columnIndex)	Retrieves the value of the specified column as a double.
double getDouble(String columnLabel)	Retrieves the value of the specified column (by name) as a double.
boolean getBoolean(int columnIndex)	Retrieves the value of the specified column as a Boolean.
boolean getBoolean(String columnLabel)	Retrieves the value of the specified column (by name) as a Boolean.
Date getDate(int columnIndex)	Retrieves the value of the specified column as a Date object.
Date getDate(String columnLabel)	Retrieves the value of the specified column (by name) as a Date object.
boolean absolute(int row)	Moves the cursor to the specified row number in the ResultSet. Returns true if the cursor is on a valid row; false if the row number is out of range.
boolean relative(int rows)	Moves the cursor a relative number of rows from the current row. Returns true if the cursor is on a valid row; false if the resulting row position is out of range.
boolean isBeforeFirst()	Returns true if the cursor is positioned before the first row; false otherwise.
boolean isAfterLast()	Returns true if the cursor is positioned after the last row; false otherwise.
boolean isFirst()	Returns true if the cursor is on the first row; false otherwise.
boolean isLast()	Returns true if the cursor is on the last row; false otherwise.
void close()	Closes this ResultSet object, releasing its resources.
int getRow()	Retrieves the current row number.
void updateString(int columnIndex, String x)	Updates the specified column with a String value.
void updateInt(int columnIndex, int x)	Updates the specified column with the given int value.
void updateDouble(int columnIndex, double x)	Updates the specified column with the given double value.
void insertRow()	Inserts the contents of the Insert row into the database.
void updateRow()	Updates the current row with the updated values.
void deleteRow()	Deletes the current row from the ResultSet and the database.
void moveToInsertRow()	Moves the cursor to the insert row, a special row used for inserting a new record.
void moveToCurrentRow()	Moves the cursor back to the current row.

***Table 10.6**: Commonly used methods of ResultSet interface*

These methods provide functionality for navigating, retrieving, and manipulating data in a ResultSet.

The following code block demonstrate the use of ResultSet object:

```
while (rs.next()) {
   int id = rs.getInt("id");
```

```
    String name = rs.getString("name");
    System.out.println("ID: " + id + ", Name: " + name);
}
```

SQLException

The SQLException class is an exception that provides information on a database access error or other errors related to JDBC operations. SQLException is used to catch and handle exceptions that occur during JDBC operations, allowing for robust error handling. It extends java.lang.Exception and can be used to handle errors related to SQL operations. Some of its commonly used methods are listed below:

Method	Description
getErrorCode()	Retrieves the error code associated with the exception, which is specific to the database.
getSQLState()	Retrieves the SQLState, a string that follows the SQL standard and provides information about the exception.
getMessage()	Retrieves a detailed message describing the exception.

***Table 10.7**: Commonly used methods of ResultSet interface*

The following code block demonstrates the usage of **SQLException**:

```
try {
    Connection conn = DriverManager.getConnection("jdbc:mysql://
localhost:3306/mydatabase", "username", "password");
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery("SELECT * FROM employees");
} catch (SQLException e) {
    System.out.println("SQL Error: " + e.getMessage());
    System.out.println("SQLState: " + e.getSQLState());
    System.out.println("Error Code: " + e.getErrorCode());
}
```

Database operations using JDBC

Database operations using JDBC involve connecting Java applications to relational databases, allowing for interaction with the database through SQL commands. In our previous section, we saw various classes and interfaces provided by JDBC APIs that enable developers to execute SQL statements, retrieve and

manipulate data, and manage database connections. The process typically starts with loading the appropriate JDBC driver and establishing a connection to the database using the DriverManager class. Once connected, developers can create and execute SQL queries using Statement, PreparedStatement, or CallableStatement objects, depending on the complexity and nature of the queries. Results from queries are handled through the ResultSet interface, which allows for navigating and processing the data retrieved from the database. Error handling in JDBC is managed using SQLException, which provides detailed information about any issues encountered during database operations. JDBC abstracts the underlying database interactions, making it easier to perform tasks like inserting, updating, deleting, and querying data while ensuring the application remains database-agnostic.

Below, we will explore the key database operations you can perform using JDBC, detailing each step and providing examples.

Loading the JDBC driver

Before interacting with a database, the JDBC driver for the specific database must be loaded. This step involves loading the driver class into memory, which registers it with the DriverManager. In modern JDBC versions (JDBC 4.0 and later), the driver is automatically loaded when the connection is established, so explicit driver loading is usually not required.

The general syntax for loading driver is as follows:

```
// Older versions required explicit loading of the driver
Class.forName("com.mysql.cj.jdbc.Driver");
```

In most cases, you can skip this step as the driver is loaded automatically.

Establishing a connection

The next step is to establish a connection to the database. This is done using the **DriverManager.getConnection()** method, which requires the database URL, username, and password. For establishing a connection with a MySQL database the statement will be as follows:

```
Connection conn = DriverManager.getConnection("jdbc:mysql://
```

localhost:3306/mydatabase", "username", "password");

In this statement, **Connection conn** is the declaration of a **Connection** object named conn. Through this connection, you can execute SQL statements and manage transactions. **conn** will hold the reference to the connection object once it is successfully established. **DriverManager.getConnection(...)** attempts to establish a connection to the database using the specified URL, username, and password. The method **getConnection** is a static method of the **DriverManager** class that attempts to establish a connection to the database specified by the URL and credentials provided. "**jdbc:mysql://localhost:3306/mydatabase**" is the JDBC URL that specifies the database to connect to:

- **jdbc:mysql**: This part specifies the protocol and the subprotocol. Here, JDBC indicates that it is a JDBC connection, and MySQL specifies that the connection is to a MySQL database.
- **//localhost:3306/**: This specifies the hostname and port number of the database server.

  localhost indicates that the database server is running on the same machine as the Java application. If the database were on a different server, you would replace localhost with the server's IP address or domain name.

  3306 is the default port number for MySQL. If MySQL is running on a different port, you would replace 3306 with the appropriate port number.
- **mydatabase**: This specifies the name of the database you want to connect to. Replace **mydatabase** with the actual name of the database you want to access.
- "**username**": It is the username required to connect to the database. Replace "**username**" with the actual username that has access rights to the specified database.
- "**password**": It is the password corresponding to the username. Replace "**password**" with the actual password for the specified username.

Creating a Statement

Once you have a **Connection** object, you can create a **Statement** object using the **createStatement()** method of the **Connection** interface. The **Statement** object allows you to send SQL statements to the database. For example:

Statement stmt = conn.createStatement();

This Statement object can now be used to execute various SQL commands like SELECT, INSERT, UPDATE, and DELETE.

Executing SQL queries

JDBC allows you to execute various types of SQL queries, including SELECT, INSERT, UPDATE, DELETE, and others. The type of query determines the method you use to execute it.

A **SELECT** query is used to retrieve data from the database. The result is returned as a **ResultSet** object.

Example:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM employees");
while (rs.next()) {
    int id = rs.getInt("id");
    String name = rs.getString("name");
    System.out.println("ID: " + id + ", Name: " + name);
}
```

An **INSERT** query adds new records to a table. The **executeUpdate()** method is used for this purpose, returning the number of affected rows.

Example:

```
int rows = stmt.executeUpdate("INSERT INTO employees (name, age,
department) VALUES ('John Doe', 30, 'HR')");
System.out.println("Rows inserted: " + rows);
```

An **UPDATE** query modifies existing records in a table. Like the **INSERT** operation, it uses the **executeUpdate()** method.

Example:

```
int rows = stmt.executeUpdate("UPDATE employees SET age = 31
WHERE name = 'John Doe'");
System.out.println("Rows updated: " + rows);
```

A **DELETE** query removes records from a table. The

executeUpdate() method is also used here.

Example:

```
int rows = stmt.executeUpdate("DELETE FROM employees WHERE
name = 'John Doe'");
System.out.println("Rows deleted: " + rows);
```

Data definition language (DDL) queries such as CREATE, ALTER, and DROP modify the structure of database objects like tables and indexes. They also use the **executeUpdate()** method.

Example:

```
stmt.executeUpdate("CREATE TABLE departments (id INT PRIMARY
KEY, name VARCHAR(50))");
```

Handling transactions

In JDBC, transactions allow multiple SQL operations to be treated as a single unit of work. Transactions ensure that either all operations are successfully executed (commit) or none of them are (rollback) if an error occurs.

By default, JDBC connections operate in auto-commit mode, meaning each SQL statement is treated as a transaction and committed immediately.

You can disable auto-commit to manually control transactions. The below listed code demonstrates how to do this:

```
conn.setAutoCommit(false); // Disable auto-commit
try {
   stmt.executeUpdate("UPDATE accounts SET balance = balance - 100
WHERE account_id = 1");
   stmt.executeUpdate("UPDATE accounts SET balance = balance +
100 WHERE account_id = 2");
   conn.commit(); // Commit transaction
} catch (SQLException e) {
   conn.rollback(); // Rollback transaction in case of error
   e.printStackTrace();
} finally {
   conn.setAutoCommit(true); // Restore auto-commit mode
}
```

Handling ResultSet

After executing a SELECT query, the result is returned as a ResultSet object. You can navigate through the ResultSet using the **next()** method and retrieve column values using the appropriate getter methods. For processing all the records in our resultset we need to use a loop. Let us see a sample code:

```
while (rs.next()) {
    int id = rs.getInt("id");
    String name = rs.getString("name");
    System.out.println("ID: " + id + ", Name: " + name);
}
```

Closing resources

It is important to close JDBC resources after they are no longer needed to free up database connections and avoid resource leaks. This includes closing the ResultSet, Statement, and Connection objects. The following code snippet demonstrates the closing of various resources:

```
rs.close();
stmt.close();
conn.close();
```

Handling exceptions

JDBC operations can throw SQLException, which needs to be handled to ensure the application can recover from database errors gracefully. The SQLException class provides methods to retrieve the error code, SQLState, and detailed error message. Let us see the sample code:

```
try {
    Connection conn = DriverManager.getConnection("jdbc:mysql://
localhost:3306/mydatabase", "username", "password");
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery("SELECT * FROM employees");
} catch (SQLException e) {
    System.out.println("SQL Error: " + e.getMessage());
    System.out.println("SQLState: " + e.getSQLState());
```

```
    System.out.println("Error Code: " + e.getErrorCode());
} finally {
    // Always close resources in the finally block to ensure they are
closed even if an exception occurs
    try { if (rs != null) rs.close(); } catch (SQLException e) {
e.printStackTrace(); }
    try { if (stmt != null) stmt.close(); } catch (SQLException e) {
e.printStackTrace(); }
    try { if (conn != null) conn.close(); } catch (SQLException e) {
e.printStackTrace(); }
}
```

Creating a simple JDBC application

Let us sum up all above-mentioned steps to build a database enabled java application. Here is a simple example of a JDBC application that connects to a MySQL database, inserts a record into a table, retrieves data, and then prints it to the console. This example assumes that a MySQL database named **testdb** is already set up with a table called employees:

```
1. import java.sql.Connection;
2. import java.sql.DriverManager;
3. import java.sql.ResultSet;
4. import java.sql.SQLException;
5. import java.sql.Statement;
6.
7. public class SimpleJDBCExample {
8.     public static void main(String[] args) {
9.         // Database URL, username, and password
10.        String url = "jdbc:mysql://localhost:3306/testdb";
11.        String user = "root";
12.        String password = "password";
13.
14.        // SQL statements
15.        String insertSQL = "INSERT INTO employees (name, role) VALUE
16.        String selectSQL = "SELECT id, name, role FROM employees";
17.
18.        // Establishing a connection
19.        try (Connection conn = DriverManager.getConnection(url, user, p
```

```
20.             Statement stmt = conn.createStatement()) {
21.
22.             // Inserting a record
23.             int rowsInserted = stmt.executeUpdate(insertSQL);
24.             System.out.println("Rows inserted: " + rowsInserted);
25.
26.             // Retrieving and displaying records
27.             ResultSet rs = stmt.executeQuery(selectSQL);
28.             while (rs.next()) {
29.                 int id = rs.getInt("id");
30.                 String name = rs.getString("name");
31.                 String role = rs.getString("role");
32.                 System.out.println("ID: " + id + ", Name: " + name + ", Rol
33.             }
34.
35.             // Closing the ResultSet
36.             rs.close();
37.
38.         } catch (SQLException e) {
39.             e.printStackTrace();
40.         }
41.     }
42. }
```

java.sql.Connection, **java.sql.DriverManager**, **java.sql.ResultSet**, **java.sql.SQLException**, and **java.sql.Statement** are imported to handle the database operations. The URL, user, and password variables store the database connection details. The URL specifies the database type (**mysql**), the host (**localhost**), and the database name (**testdb**). insertSQL is the SQL command to insert a new employee record into the employees table. selectSQL is the SQL command to retrieve all records from the employees table. The DriverManager.getConnection(url, user, password) method establishes a connection to the database using the specified URL, username, and password. **stmt.executeUpdate(insertSQL)** executes the INSERT statement, adding a new employee record. The number of affected rows is returned and printed. **stmt.executeQuery(selectSQL)** executes the SELECT statement and returns the result in a ResultSet. The ResultSet is iterated using a

while loop, retrieving each record's id, name, and role and printing them. The entire database operation is enclosed in a try block, with catch handling any SQLException that might occur. The ResultSet is explicitly closed after processing. The Connection and Statement are closed automatically because they are declared within the try block using the try-with-resources statement.

When you run this program, it will insert a new employee named **Alice** with the role **Engineer** into the employees table and then retrieve and display all records in the table.

Rows inserted: 1
ID: 1, Name: Alice, Role: Engineer

Note: The output assumes that the employees table is empty before running the program. If the table already has records, those records will also be displayed. The table employees are expected to have at least three columns: id (auto-incremented primary key), name (VARCHAR), and role (VARCHAR).

Replace **password** in the connection string with the actual password for your MySQL root user or another database user with appropriate privileges.

Using prepared statement

PreparedStatement is used in scenarios where an application needs to execute the same SQL query multiple times with different parameters, especially in dynamic and secure database operations. A common use case is when inserting a large number of records into a database, such as in batch processing or bulk data imports. The PreparedStatement allows the SQL query to be precompiled by the database, which not only improves performance but also enhances security by preventing SQL injection attacks.

For example, consider a scenario where an application needs to store user information in a database. Using PreparedStatement, the SQL query is written with placeholders for the user data. Here is a program demonstrating the use of PreparedStatement for inserting and retrieving user data from a database:

```
1. import java.sql.Connection;
2. import java.sql.DriverManager;
3. import java.sql.PreparedStatement;
```

```
4. import java.sql.ResultSet;
5. import java.sql.SQLException;
6.
7. public class PreparedStatementExample {
8.     public static void main(String[] args) {
9.         // Database connection details
10.        String url = "jdbc:mysql://localhost:3306/testdb";
11.        String user = "root";
12.        String password = "password";
13.
14.        // SQL query with placeholders for inserting data
15.        String insertSQL = "INSERT INTO users (username, email, passwo
16.        String selectSQL = "SELECT id, username, email FROM users";
17.
18.        try (
19.            // Establish the database connection
20.            Connection conn = DriverManager.getConnection(url, user, pas
21.
22.            // Create the PreparedStatement for inserting data
23.            PreparedStatement pstmt = conn.prepareStatement(insertSQL);
24.
25.            // Create the PreparedStatement for selecting data
26.            PreparedStatement selectPstmt = conn.prepareStatement(selec
27.        ) {
28.            // Insert first user
29.            pstmt.setString(1, "john_doe");
30.            pstmt.setString(2, "john@example.com");
31.            pstmt.setString(3, "securepassword");
32.            pstmt.executeUpdate();
33.
34.            // Insert second user
35.            pstmt.setString(1, "jane_smith");
36.            pstmt.setString(2, "jane@example.com");
37.            pstmt.setString(3, "anothersecurepassword");
38.            pstmt.executeUpdate();
39.
40.            // Execute the SELECT statement to retrieve data
41.            ResultSet rs = selectPstmt.executeQuery();
42.
43.            // Display the results
```

```
            while (rs.next()) {
                int id = rs.getInt("id");
                String username = rs.getString("username");
                String email = rs.getString("email");
                System.out.println("ID: " + id + ", Username: " + username
            }

            // Close the ResultSet
            rs.close();

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

In this example program, the **insertSQL** string contains an SQL query with placeholders (?) for inserting user data (username, email, password). The program uses **PreparedStatement** to insert two users into the users table. The **setString()** method is used to set the values for each placeholder in the query. After inserting the users, a second **PreparedStatement** is used to retrieve and display the data from the users table. The program prints the id, username, and email of all users currently in the users table:

ID: 1, Username: john_doe, Email: john@example.com
ID: 2, Username: jane_smith, Email: jane@example.com

Notes: Ensure that the testdb database and users table exist in your MySQL database. The users table should have columns id (auto-increment), username, email, and password. If these tables do not exist, you can create the users table with the following SQL command:

```
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) NOT NULL,
    email VARCHAR(50) NOT NULL,
    password VARCHAR(50) NOT NULL
);
```

Using CallableStatement

CallableStatement is used in JDBC to execute stored procedures in a database. Stored procedures are precompiled SQL code that can perform complex operations like data manipulation, business logic, or batch processing. Using CallableStatement is particularly useful when the same operation needs to be performed repeatedly with different parameters, or when the logic is too complex to be handled efficiently with standard SQL queries.

Imagine a scenario where you need to retrieve the details of an employee based on their ID and calculate their yearly bonus based on predefined business logic stored in the database. Instead of writing complex SQL logic in your Java application, you can leverage a stored procedure to encapsulate this logic.

Suppose the database has the following stored procedure:

```
DELIMITER //
CREATE PROCEDURE GetEmployeeBonus(IN emp_id INT, OUT
emp_name VARCHAR(50), OUT bonus DOUBLE)
BEGIN
    DECLARE salary DOUBLE;
   -- Retrieve employee's name and salary based on emp_id
    SELECT name, salary INTO emp_name, salary FROM employees
WHERE id = emp_id;
    -- Calculate bonus as 10% of salary
    SET bonus = salary * 0.10;
END //
DELIMITER ;
```

This stored procedure **GetEmployeeBonus** takes an employee's ID as input, retrieves their name and salary, and calculates a bonus (10% of the salary). The procedure returns the employee's name and calculated bonus. Let us see Java code using **CallableStatement**:

```
1. import java.sql.Connection;
2. import java.sql.DriverManager;
3. import java.sql.CallableStatement;
4. import java.sql.SQLException;
5.
6. public class CallableStatementExample {
7.     public static void main(String[] args) {
8.         // Database connection details
```

```
9.      String url = "jdbc:mysql://localhost:3306/testdb";
10.     String user = "root";
11.     String password = "password";
12.
13.     // JDBC objects
14.     Connection conn = null;
15.     CallableStatement cstmt = null;
16.
17.     try {
18.         // Establish the connection
19.         conn = DriverManager.getConnection(url, user, password);
20.
21.         // Prepare the CallableStatement for the stored procedure
22.         cstmt = conn.prepareCall("{call GetEmployeeBonus(?, ?, ?)}");
23.
24.         // Set the input parameter (employee ID)
25.         cstmt.setInt(1, 101);
26.
27.         // Register the output parameters
28.         cstmt.registerOutParameter(2, java.sql.Types.VARCHAR);
29.         cstmt.registerOutParameter(3, java.sql.Types.DOUBLE);
30.
31.         // Execute the stored procedure
32.         cstmt.execute();
33.
34.         // Retrieve the output parameters
35.         String empName = cstmt.getString(2);
36.         double bonus = cstmt.getDouble(3);
37.
38.         // Display the result
39.         System.out.println("Employee Name: " + empName);
40.         System.out.println("Calculated Bonus: $" + bonus);
41.
42.     } catch (SQLException e) {
43.         e.printStackTrace();
44.     } finally {
45.         try {
46.             if (cstmt != null) cstmt.close();
47.             if (conn != null) conn.close();
```

```
48.             } catch (SQLException e) {
49.                 e.printStackTrace();
50.             }
51.         }
52.     }
53. }
```

The **CallableStatement** object is created using the **Connection.prepareCall()** method, with the stored procedure's call syntax **{call GetEmployeeBonus(?, ?, ?)}**. The employee ID is set as the input parameter using **setInt(1, 101)**. The output parameters (**emp_nam** and bonus) are registered using **registerOutParameter()** methods. The **execute()** method is called to run the stored procedure. The output values are retrieved using **getString(2)** and **getDouble(3)** for the employee's name and bonus, respectively. The calculated bonus and employee name are printed to the console. Output will be:

Employee Name: John Doe
Calculated Bonus: $5000.0

Exploring ResultSet operations

A ResultSet object in JDBC functions similarly to a 2D array, where each call to **next()** advances the cursor to the next record in the result set. You need to call **next()** before accessing the first record, and it returns false when no more records are available, making it convenient for controlling loops. The ResultSet class provides various getter methods like **getBlob()**, **getBigDecimal()**, **getDate()**, **getBytes()**, **getInt()**, **getLong()**, **getString()**, and **getObject()** to retrieve SQL data types by column name or number. By default, a ResultSet is not updatable and only supports forward movement of the cursor. However, many database drivers offer support for scrollable and updatable ResultSet objects. Scrollable result sets allow the cursor to move both backward and forward through records, while updatable result sets enable modification of the data and updating the database with these changes. Note that not all updates in an updatable result set can be reflected in the database, as this depends on the complexity of the query and the derivation of the data in the result set.

In JDBC, ResultSet constants are used to specify certain properties and behaviors of a ResultSet object when it is created. These constants are typically used when creating a Statement or PreparedStatement to define how the ResultSet should behave in terms of cursor movement, concurrency, and holdability. Here is an explanation of the key ResultSet constants:

- **ResultSet type constants**: These constants determine how the cursor can navigate through the rows of the ResultSet. There are three main type constants in the ResultSet interface:
- **TYPE_FORWARD_ONLY**: The cursor can only move forward through the result set. This is the default type if no specific type is set. This type is typically used for simple queries where you only need to process each row once.
- **TYPE_SCROLL_INSENSITIVE**: The cursor can move both forward and backward, but the result set is not sensitive to changes made to the underlying data by others while it is open. In other words, the data seen by the ResultSet remains static and does not reflect changes made by other transactions.
- **TYPE_SCROLL_SENSITIVE**: The cursor can move forward and backward, and the ResultSet is sensitive to changes in the underlying data. This means if the data in the database changes after the ResultSet is opened, those changes may be reflected in the ResultSet.

ResultSet concurrency constants

In JDBC, ResultSet concurrency constants define how you can interact with the data in a ResultSet. Specifically, they determine whether you can update the data in the ResultSet and how those updates are managed:

- **CONCUR_READ_ONLY**: The ResultSet is read-only. You cannot use this ResultSet to update the data in the database. This is the default concurrency mode.
- **CONCUR_UPDATABLE**: The ResultSet is updatable. You can modify the data in the ResultSet, and these changes can be propagated back to the database.

ResultSet holdability constants

ResultSet holdability constants determine how long a ResultSet should remain open and available after the Statement that created it has been closed. Holdability is an important concept in database applications because it affects how you manage resources and handle ResultSet objects in your code. These constants specify what happens to the ResultSet when the transaction that created it is committed.

- **HOLD_CURSORS_OVER_COMMIT**: The cursor is not closed when the transaction is committed. This allows the ResultSet to remain open across commits, which can be useful in long-running applications.
- **CLOSE_CURSORS_AT_COMMIT**: The cursor is automatically closed when the transaction that created it is committed. This is the default behavior, ensuring that resources are freed up once the transaction is complete.

The following code snippets demonstrate using these constants:

When creating a **Statement** or **PreparedStatement**, you can specify these properties as follows:

```
Statement stmt = conn.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_UPDATABLE
);
ResultSet rs = stmt.executeQuery("SELECT * FROM employees");
```

In this example, the ResultSet will be scrollable (allowing movement in both directions) and updatable, but it will not reflect changes made by other transactions to the underlying data.

JDBC with Swing

In the previous chapter, we learned how to create GUI using swing. Here is a complete example of a Java Swing application that creates a data entry form for student data and connects to a database using JDBC. The form allows users to enter student information (e.g., ID, name, and age) and submit this data to a MySQL database:

```
1. import javax.swing.*;
2. import java.awt.*;
```

```
3. import java.awt.event.ActionEvent;
4. import java.awt.event.ActionListener;
5. import java.sql.Connection;
6. import java.sql.DriverManager;
7. import java.sql.PreparedStatement;
8. import java.sql.SQLException;
9.
10. public class StudentEntryForm {
11.
12.     public static void main(String[] args) {
13.         // Create the frame
14.         JFrame frame = new JFrame("Student Data Entry");
15.         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
16.         frame.setSize(400, 300);
17.         frame.setLayout(new GridLayout(5, 2, 10, 10));
18.
19.         // Create form labels and fields
20.         JLabel idLabel = new JLabel("Student ID:");
21.         JTextField idField = new JTextField();
22.
23.         JLabel nameLabel = new JLabel("Name:");
24.         JTextField nameField = new JTextField();
25.
26.         JLabel ageLabel = new JLabel("Age:");
27.         JTextField ageField = new JTextField();
28.
29.         JButton submitButton = new JButton("Submit");
30.
31.         // Add components to the frame
32.         frame.add(idLabel);
33.         frame.add(idField);
34.
35.         frame.add(nameLabel);
36.         frame.add(nameField);
37.
38.         frame.add(ageLabel);
39.         frame.add(ageField);
40.
41.         frame.add(new JLabel()); // Empty cell for alignment
```

```
42.         frame.add(submitButton);
43.
44.         // Define the action for the submit button
45.         submitButton.addActionListener(new ActionListener() {
46.             @Override
47.             public void actionPerformed(ActionEvent e) {
48.                 String idStr = idField.getText();
49.                 String name = nameField.getText();
50.                 String ageStr = ageField.getText();
51.
52.                 // Basic validation
53.                 if (idStr.isEmpty() || name.isEmpty() ||
    ageStr.isEmpty()) {
54.                     JOptionPane.showMessageDialog(frame, "All fields are req
55.                     return;
56.                 }
57.
58.                 try {
59.                     int id = Integer.parseInt(idStr);
60.                     int age = Integer.parseInt(ageStr);
61.
62.                     // JDBC connection details
63.                     String url = "jdbc:mysql://localhost:3306/
    studentdb";
64.                     String user = "root"; // Replace with your MySQL usernan
65.                     String password = "password"; // Replace with your MyS
66.
67.                     // Establish the connection
68.                     Connection connection = DriverManager.getConnection(u
69.
70.                     // Prepare the SQL statement
71.                     String sql = "INSERT INTO students (id, name, age) VALU
72.                     PreparedStatement preparedStatement = connection.prep
73.                     preparedStatement.setInt(1, id);
74.                     preparedStatement.setString(2, name);
75.                     preparedStatement.setInt(3, age);
76.
77.                     // Execute the update
78.                     int rowsAffected = preparedStatement.executeUpdate();
79.                     JOptionPane.showMessageDialog(frame, "Student data sub
```

```
\nRows affected: " + rowsAffected, "Submission Successful", JOptionPa
80.
81.                 // Close resources
82.                 preparedStatement.close();
83.                 connection.close();
84.
85.             } catch (NumberFormatException ex) {
86.                 JOptionPane.showMessageDialog(frame, "ID and Age must
87.             } catch (SQLException ex) {
88.                 ex.printStackTrace();
89.                 JOptionPane.showMessageDialog(frame, "Database error o
90.             }
91.         }
92.     });
93.
94.     // Set the frame visibility
95.     frame.setVisible(true);
96.   }
97. }
```

Prerequisites for this application are as follows. Create a MySQL database and a table to store student data:

```
CREATE DATABASE studentdb;
USE studentdb;
CREATE TABLE students (
   id INT PRIMARY KEY,
   name VARCHAR(100),
   age INT
);
```

This example provides a straightforward way to create a data entry form for student information and connect it to a MySQL database using Swing and JDBC. Adjust the database URL, userzname, and password as needed for your environment.

Mapping of Java-SQL data types

When working with JDBC, it is crucial to understand how Java data types map to SQL data types. This mapping ensures that the data can be correctly transferred between a Java application and a database. Here is a summary of the common mappings between

Java and SQL data types:

	SQL data type	
[illegible]	[illegible]	string types in SQL. Used for textual data.
[illegible]	[illegible]	error false value.
[illegible]	[illegible]	small integer value.
[illegible]	[illegible]	a small integer.
[illegible]	[illegible]	integer mapping.
[illegible]	[illegible]	a large integer value.
[illegible]	[illegible]	cision floating-point number.
[illegible]	[illegible]	floating-point number.
[illegible]	[illegible]	cision numeric values, such as monetary amounts.
[illegible]	[illegible]	ages or files.
[illegible]	[illegible]	esents a date (year, month, day).
[illegible]	[illegible]	esents a time (hour, minute, second).
[illegible]	[illegible]	ne.
[illegible]	[illegible]	SQL array types.
[illegible]	[illegible]	binary large objects, such as images or files.
[illegible]	[illegible]	character large objects, often used for storing large text data.
[illegible]	[illegible]	structured types (used for user-defined types in the database).
[illegible]	[illegible]	a reference to an SQL structured type.
[illegible]	[illegible]	database-specific row identifier.
[illegible]	[illegible]	data.

***Table 10.8**: Mapping of Java-SQL data types*

- **Character and string types**: In Java, String corresponds to various SQL types that store character data, such as CHAR, VARCHAR, TEXT, and CLOB.
- **Numeric types**: Java numeric types like int, long, float, and double map to SQL numeric types like INTEGER, BIGINT, REAL, and DOUBLE. BigDecimal in Java is used when precision is critical, and it maps to SQL DECIMAL or NUMERIC.
- **Binary data**: byte[] in Java maps to SQL types like BLOB, BINARY, or VARBINARY, which are used to store binary data like images or files.
- **Date and time types**: Java's Date, Time, and Timestamp map to SQL DATE, TIME, and TIMESTAMP types respectively, handling different aspects of date and time data.
- **Specialized types**: SQL also has specialized types like

ARRAY, STRUCT, and XML, which have corresponding classes in the **java.sql** package (e.g., **java.sql.Array**, **java.sql.Struct**, **java.sql.SQLXML**).

Using advanced data types

In JDBC, **Binary Large Object (BLOB)** and **Character Large Object (CLOB)** are used to handle large amounts of data stored in a database. They allow you to store and retrieve large binary or text data, respectively.

Binary Large Object

BLOB is used to store large binary data, such as images, multimedia files, or other types of non-textual data. It is useful when you need to handle large volumes of binary data in a database. Common BLOB Methods are listed below:

- **getBlob(String columnLabel)**: Retrieves the BLOB value for the specified column label.
- **setBlob(int parameterIndex, Blob x)**: Sets the parameter to a Blob object in a PreparedStatement.

Let us discuss one example:

```
1. import java.io.File;
2. import java.io.FileInputStream;
3. import java.io.IOException;
4. import java.sql.Connection;
5. import java.sql.DriverManager;
6. import java.sql.PreparedStatement;
7. import java.sql.SQLException;
8.
9. public class BlobInsertExample {
10.     public static void main(String[] args) {
11.         Connection connection = null;
12.         PreparedStatement preparedStatement = null;
13.         FileInputStream fileInputStream = null;
14.
15.         try {
16.             // Load the JDBC driver
17.             Class.forName("com.mysql.cj.jdbc.Driver");
```

```
18.
19.             // Establish the connection
20.             connection = DriverManager.getConnection("jdbc:mysql://
    localhost:3306/yourdatabase", "username", "password");
21.
22.             // Specify the file to be inserted
23.             File imageFile = new File("C:/Image.jpg");
24.             fileInputStream = new FileInputStream(imageFile);
25.
26.             // Prepare the SQL statement
27.             String sql = "INSERT INTO blobtest (pic_name, pic_file) VALUE
28.             preparedStatement = connection.prepareStatement(sql);
29.             preparedStatement.setString(1, imageFile.getName());
30.             preparedStatement.setBinaryStream(2, fileInputStream, (int) im
31.
32.             // Execute the update
33.             int rowsAffected = preparedStatement.executeUpdate();
34.             System.out.println("Rows affected: " + rowsAffected);
35.
36.         } catch (ClassNotFoundException | SQLException |
     IOException e) {
37.             e.printStackTrace();
38.         } finally {
39.             // Close resources
40.             try {
41.                 if (preparedStatement !
    = null) preparedStatement.close();
42.                 if (connection != null) connection.close();
43.                 if (fileInputStream != null) fileInputStream.close();
44.             } catch (SQLException | IOException e) {
45.                 e.printStackTrace();
46.             }
47.         }
48.     }
49. }
```

The provided Java code demonstrates how to insert binary data into a MySQL database using JDBC. A **FileInputStream** is used to read a binary file (e.g., an image) from the filesystem. The code then prepares an SQL INSERT statement with placeholders for the file name and the binary data, which are set using **PreparedStatement**.

After executing the statement, it prints the number of rows affected. The code includes proper resource management by closing the **PreparedStatement**, **Connection**, and **FileInputStream** in a finally block to ensure resources are released even if an exception occurs.

Character Large Object

Character Large Object (CLOB) is used to store large text data, such as long documents or large amounts of text. It is useful for handling textual data that exceeds the size of standard VARCHAR fields. Common CLOB methods are as follows:

- **getClob(String columnLabel)**: Retrieves the CLOB value for the specified column label.
- **setClob(int parameterIndex, Clob x)**: Sets the parameter to a **Clob** object in a **PreparedStatement**.

Let us see one example:

```
1. import java.sql.*;
2.
3. public class ClobUpdateExample {
4.     public static void main(String[] args) {
5.         Connection connection = null;
6.         PreparedStatement selectStmt = null;
7.         PreparedStatement updateStmt = null;
8.         ResultSet resultSet = null;
9.         Clob notesClob = null;
10.
11.        try {
12.            // Load the JDBC driver
13.            Class.forName("com.mysql.cj.jdbc.Driver");
14.
15.            // Establish the connection
16.            connection = DriverManager.getConnection("jdbc:mysql://
    localhost:3306/yourdatabase", "username", "password");
17.
18.            // Retrieve the CLOB data
19.            String selectSQL = "SELECT NOTES FROM your_table WHERE s
20.            selectStmt = connection.prepareStatement(selectSQL);
21.            resultSet = selectStmt.executeQuery();
22.
```

```
            if (resultSet.next()) {
                notesClob = resultSet.getClob("NOTES");
            }

            // Update the COMMENTS column using the retrieved CLOB
            String updateSQL = "UPDATE MARKETS SET COMMENTS = ?
            updateStmt = connection.prepareStatement(updateSQL);
            if (notesClob != null) {
                updateStmt.setClob(1, notesClob);
                int rowsAffected = updateStmt.executeUpdate();
                System.out.println("Rows updated: " + rowsAffected);
            } else {
                System.out.println("No CLOB data found to update.");
            }

        } catch (ClassNotFoundException | SQLException e) {
            e.printStackTrace();
        } finally {
            // Close resources
            try {
                if (resultSet != null) resultSet.close();
                if (selectStmt != null) selectStmt.close();
                if (updateStmt != null) updateStmt.close();
                if (connection != null) connection.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

This Java program demonstrates how to handle CLOB data with JDBC. It starts by establishing a connection to a MySQL database and selecting a CLOB value from a specific table based on a condition. The CLOB data is retrieved using a PreparedStatement and ResultSet. Next, the program prepares an UPDATE statement to modify the COMMENTS column in the MARKETS table where sales are less than 1,000,000. The retrieved CLOB data is set as the parameter for the update query and executed. Proper resource management is ensured by closing the ResultSet,

PreparedStatement, and Connection in a finally block, making the program robust and preventing resource leaks.

Conclusion

In this chapter, we have comprehensively covered the fundamental aspects of JDBC, providing a solid foundation for working with databases in Java applications. We began with an exploration of JDBC architecture, understanding its key components and how they facilitate database interactions. By examining the different types of JDBC drivers, we identified how each type serves distinct purposes and scenarios, ensuring optimal connectivity between Java applications and databases.

We then discussed the major JDBC classes and interfaces, such as DriverManager, Connection, Statement, and ResultSet, which are essential for performing database operations. Through practical examples, we demonstrated how to execute queries, handle results, and manage database connections effectively.

Our exploration of JDBC included creating a simple JDBC application, which illustrated the process of connecting to a database, executing SQL commands, and processing results. We also detailed the different types of statements—Statement, PreparedStatement, and CallableStatement—highlighting their specific use cases and advantages in preparing and executing SQL statements. Additionally, we examined various ResultSet operations, showing how to navigate and manipulate query results efficiently. By understanding these operations, developers can retrieve and process data from the database with precision.

Overall, this chapter has provided a comprehensive overview of JDBC, equipping you with the knowledge to create robust, data-driven applications. The concepts and examples covered lay the groundwork for developing more advanced database applications and integrating complex data management features into your Java projects.

In the next chapter, we will learn networking concept in Java.

Exercises

1. Write a Java program to connect to a MySQL database and execute a simple SQL query to retrieve all records from a table named employees. Display the results in the console.
2. Create a Java program that inserts a new record into a students table with columns id, name, and age. Prompt the user to enter values for these columns and use PreparedStatement to insert the data.
3. Write a Java program to update the salary of an employee in the employees table based on the employee ID provided by the user. Use PreparedStatement for the update operation.
4. Implement a Java program that deletes a record from the employees table based on the employee ID provided by the user. Use PreparedStatement to perform the deletion.
5. Write a Java program to retrieve all records from a products table and display the data in a graphical JTable using Swing. The table should have columns for product_id, product_name, and price.
6. Create a Java program that performs batch inserts into a orders table. Insert multiple records in a single transaction using PreparedStatement and batch processing.
7. Write a Java program that calls a stored procedure from the database. The stored procedure should accept parameters and return a result set. Assume a stored procedure named getEmployeeDetails exists in the database.
8. Create a Java program that performs multiple database operations within a single transaction. Implement commit and rollback functionality to ensure data consistency.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 11
Advanced Networking in Java

Introduction

Network programming refers to writing programs that can work over network. Computers on the Internet communicate using either the **Transmission Control Protocol (TCP)** or the **User Datagram Protocol (UDP)**. When developing Java programs that interact over a network, you are working at the application layer. This chapter introduces the fundamental concepts and tools necessary for network programming in Java, essential for developing applications that communicate over the internet or local networks. We begin with an overview of networking basics, covering key terms like IP addresses, ports, and protocols. The chapter then explores sockets, the primary endpoints for network communication, and delves into the Socket API, which provides the essential methods for managing these connections. We also examine how to interact with web resources using URLs and URLConnection, followed by a discussion on creating client-server applications using TCP/IP sockets. Finally, the chapter introduces datagrams for connectionless communication, highlighting their usage in scenarios where

message delivery order is not guaranteed. The java.net package provides classes and interfaces that enable platform-independent network communication.

Structure

The chapter covers the following topics:

- Introduction to networking basics
- Understanding socket
- Mechanism of socket
- Socket API
- Working with URLs and URLConnection
- InetAddress class
- TCP/IP client sockets and server sockets
- Datagram communication with UDP
- Example programs

Objectives

The objective of this chapter is to build a solid understanding of network programming concepts in Java, enabling you to develop applications that effectively communicate over networks. By the end of this chapter, the learner will gain knowledge of fundamental networking concepts, including IP addresses, ports, and protocols. Readers will acquire the skills to create and manage network connections using sockets, both for client-server models and peer-to-peer communication. This chapter also aims to familiarize the reader with the Java Socket API, including key classes and methods essential for network communication. It also discusses interacting with Web Resources using URLs and URLConnection to connect to web servers and retrieve data. Students will also learn to develop client-server applications using TCP/IP sockets, understanding the principles of reliable data transfer, as well as understanding the concept of datagramse4 and how to use them for connectionless communication, suitable for applications where delivery order and reliability are less critical.

Introduction to networking basics

To build a foundation for learning network programming, it is essential to understand key networking fundamentals. These concepts provide the basis for how data is transmitted, received, and managed across networks.

Network is a collection of interconnected devices (computers, servers, routers, etc.) that communicate with each other to share resources and information. A network is made up of many layers. TCP/IP model represents a network consisting of four layers. The following figure shows the layers of the network with major protocols used at the specific layer:

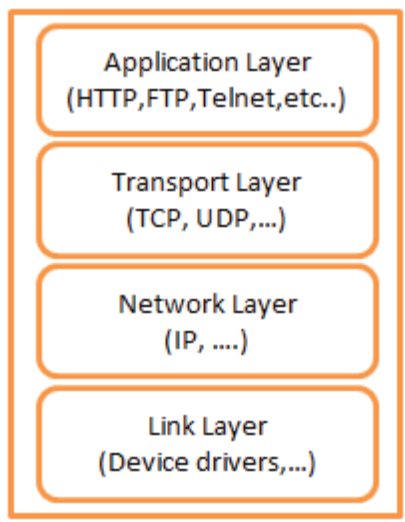


***Figure 11.1**: TCP/IP network model*

When you write Java programs to communicate over the network, you are programming at the application layer. Typically, you do not need to bother about the TCP and UDP layers. Instead, you can use the classes in the **java.net** package. Some basic concepts related to network programming are discussed below.

IP address

Data is transmitted between computers in packets. Each packet is marked with a destination address, written as an IP address and

port number. Each host over the network must be identified with a unique 32-bit IP address, which is used by internet protocol to deliver the packets to the right destination. IP addresses are often represented as a quad of four 8-bit numbers separated by periods. They are organized into classes (A, B, C, D, and E), which are used to group varying numbers of hosts in a hierarchical numbering scheme.

Internet addresses are manipulated in Java by the use of the InetAddress class. InetAddress takes care of the **Domain Name System** (**DNS**) look-up and reverse look-up; IP addresses can be specified by either the host name or the raw IP address. InetAddress provides methods to getByName(), getAllByName(), getLocalHost(), getAddress(), etc.

Understanding ports

While an IP address identifies a device on a network, a port number identifies a specific process or service on that device. Think of it as an extension number in a phone system. If the IP address is the phone number, the port number tells you which department or person to reach within that organization. Each socket on your computer is associated with a port number. Ports are identified by a 16-bit number, which TCP and UDP use to deliver the data to the right application. When a server wants to share information, it listens on a specific port. For example, when you visit a website, your web browser connects to port 80 or 443 on the server where the website is hosted. The server waits (or listens) on this port for incoming connections from clients (like your web browser). Port numbers are 16-bit numbers, which means they range from 0 to 65,535.

The port numbers between 0-1,023 are reserved for common services and protocols like HTTP (port 80), HTTPS (port 443), FTP (port 21), and others. Port numbers of the range 1024-49151 are registered ports, assigned for specific applications by *Internet Assigned Numbers Authority* (*IANA*). The range 49152-65535 is dynamic/private ports used by applications to establish temporary connections.

A server can handle many clients at once, even if they are all connecting to the same port. Each client connection is unique,

meaning the server can keep track of multiple conversations simultaneously. To manage all these connections, the server often uses multithreading—this means the server can do several tasks at once, like talking to different clients without getting confused.

Client/server computing

You can use the Java language to communicate with remote file systems using a client/server model. A server listens for connection requests from clients across the network or even from the same machine. Clients know how to connect to the server via an IP address and port number. Upon connection, the server reads the request sent by the client and responds appropriately. In this way, applications can be broken down into specific tasks that are accomplished in separate locations. The data that is sent back and forth over a socket can be anything you like. Normally, the client sends a request for information or processing to the server, which performs a task or sends data back. You could, for example, place an SQL shell on the server and let people talk to it via a simple client chat program. The IP and port number of the server are generally well-known and advertised, so the client knows where to find the service. In contrast, the port number on the client side is generally allocated automatically by the kernel. Many protocols used on the Internet (HTTP, for example) are designed to be driven from the command line. They send their requests and responses across the net in plain text. One of the easiest ways to become familiar with network programming and/or specific protocols is to use the TELNET application to talk directly to a server from the command line.

Understanding socket

The core of Java's networking capabilities revolves around the concept of a socket. A socket is like a door to a network, allowing a computer to communicate with other computers. This idea was first introduced in the early 1980s with a version of UNIX called 4.2BSD, and it is so central to how computers talk to each other today that it is sometimes called the Berkeley socket. Think of a socket as an endpoint in a network communication process. It is where data enters or leaves your computer when you are connected to a

network. Sockets are crucial because they let a single computer handle many different connections at the same time. For instance, your computer can be browsing the web, streaming music, and downloading files all at once, thanks to sockets.

A socket is an endpoint in a network connection, and it provides a communication mechanism between two machines over a network. When two programs (often on different devices) want to communicate with each other, each program creates a socket. The sockets can then connect to each other, allowing data to be sent and received.

Mechanism of socket communication

When sockets communicate, they use protocols. Protocols are sets of rules that determine how data is sent and received over a network:

- **Internet protocol (IP)**: Internet protocol provides a delivery mechanism. This is a basic, low-level protocol that breaks your data into small packets and sends them across the network. The internet protocol, commonly known as IP, is a set of rules that governs how data is sent and received over the Internet. IP is responsible for routing packets of data from one computer to another, ensuring that they reach their intended destination.
- **TCP**: TCP works on top of IP and ensures that all those packets sent by IP get to where they are supposed to go. TCP keeps track of the order of the packets and resends any that get lost, making sure your data arrives intact and in the right order.
- **UDP**: UDP is another protocol that sits alongside TCP. It is faster but less reliable because it does not check if all the packets arrived or if they are in the correct order. This is useful for things like live video or audio streaming, where speed is more important than perfect accuracy.

Socket API

The Socket **application programming interface** (**API**) consists of functions and protocols used in network programming to facilitate communication between devices over a network. It forms the

backbone of network programming, allowing the creation of applications that can communicate over the Internet or within a local network. In networking, communication happens between pairs of sockets: a server socket listens for incoming connections from clients, while a client socket initiates a connection to a server.

Java programs can use classes from the java.net package to communicate over the Internet using either TCP or UDP. The URL, URLConnection, Socket, and ServerSocket classes support communication over the network using TCP. For UDP communication, Java provides the DatagramPacket, DatagramSocket, and MulticastSocket classes.

Socket types

There are two main types of sockets based on the communication protocol they use: TCP sockets and UDP sockets.

- **TCP sockets**: TCP sockets are also known as Stream Sockets. These use the TCP and provide reliable, connection-oriented communication. Data is guaranteed to be delivered in the order it was sent, or an error is reported. TCP is a connection-oriented protocol, which means it establishes a reliable communication channel between the client and server before data transfer. Java.net package provides Socket class to be used by the client to establish a connection to the server and ServerSocket class to be used by the server to listen for incoming connection requests from clients.
- **UDP**: UDP sockets are also called datagram sockets. These use the UDP and provide connectionless communication. Data is sent as individual packets (datagrams), and delivery is not guaranteed, nor is the order of the packets. UDP is a connectionless protocol, meaning that it doesn't establish a connection before sending data. It is faster than TCP but does not guarantee reliable data transmission or ensure that data will arrive in order. Java.net package provides DatagramSocket class, which is used for sending and receiving datagram packets, and DatagramPacket class to represent the data being sent or received in a UDP packet.

  TCP Sockets are used in scenarios where data integrity and

order are crucial, such as file transfers, web communications, and email.

UDP Sockets are preferred for applications where speed is more important than reliability, such as streaming services, online games, and **voice over IP** (**VoIP**).

Both TCP and UDP have their own advantages depending on the application's requirements, and Java provides robust support for both through its socket APIs.

Socket address

A socket address uniquely identifies a communication endpoint on a network and is composed of two main components: an IP address and a port number. The IP address, which can be in IPv4 or IPv6 format, represents the unique identifier of a device or host on the network, while the port number specifies the communication endpoint for a specific application or service on that device. For instance, 192.168.1.1:80 refers to an HTTP service on IP address 192.168.1.1 at port 80. Port numbers are categorized into well-known ports (0-1023) for common services like HTTP (80) and HTTPS (443), registered ports (1024-49151) for user-defined services, and dynamic or private ports (49152-65535) for temporary connections. Together, the IP address and port ensure unique identification and enable multitasking by allowing multiple services to operate on the same device without conflict. When a client connects to a server, it specifies the server's socket address, enabling the server to listen on the specified IP and port to establish a communication channel. Socket addresses are fundamental to networking, enabling seamless data exchange in protocols like TCP/IP and UDP.

Working with URL and URLConnection

If you have used the web, you are probably familiar with the term URL and have used it to access web pages. While it is often easiest—though not entirely accurate—to think of a URL as the name of a file on the World Wide Web, most URLs indeed refer to files on networked machines. However, URLs can also point to other types of resources on the network, such as database queries or command

outputs.

A **Uniform Resource Locator (URL)** is an address used to reference a resource on the Internet. Just as you provide addresses on letters for the post office to deliver them to the right person, you supply URLs to your web browser to locate files on the Internet. Similarly, Java programs that interact with the Internet use URLs to access the resources they need. In Java, the URL class from the java.net package is used to represent a URL address.

The term URL can sometimes be ambiguous. It can either refer to an Internet address or to a URL object within a Java program. To avoid confusion, this text uses URL address when referring to an Internet address and URL object when discussing an instance of the URL class in a Java program.

A URL consists of two primary components:

- **Protocol identifier**: In the URL http://example.com, the protocol identifier is http.
- **Resource name**: In the URL http://example.com, the resource name is example.com.

These two components are separated by a colon and two forward slashes. The protocol identifier specifies the protocol used to access the resource. In this case, it is the **Hypertext Transfer Protocol (HTTP)**, commonly used to serve web pages. HTTP is just one of many protocols used to access different types of resources online. Others include FTP, Gopher, File, and News.

The resource name represents the complete address of the resource, and its format depends on the protocol being used. For many protocols, including HTTP, the resource name may include the following components:

- **Host name**: The name of the server where the resource resides.
- **Filename**: The path to the file on that server.
- **Port number**: The port to connect to (optional in most cases).
- **Reference**: A reference to a specific part within a resource, usually pinpointing a location within a file (also optional).

In many protocols, the host name and filename are mandatory, while the port number and reference are optional. For example, an HTTP URL must specify both the server (host name) and the file path (filename). Optionally, it can also include a port number and a reference.

In Java, working with URLs involves the use of the java.net package, specifically the URL and URLConnection classes, to access resources over the internet, such as websites, files, or services. These classes allow you to connect to a remote server via HTTP or other protocols, retrieve data, and interact with web resources in a high-level way.

Creating a URL

The simplest way to create a URL object is by using a String that represents the human-readable form of the URL address, which is typically how users interact with URLs. In Java, you can create a URL object from such a string as follows:

```
URL myURL = new URL("http://example.com/");
```

This creates a URL object representing an absolute URL, which contains all the information necessary to access the specified resource. You can also create URL objects from relative URLs.

The URL class offers additional constructors for creating URL objects, which are useful when working with URLs that include components like host name, filename, port number, and reference (often in HTTP URLs). These constructors are helpful when you do not have a complete URL as a string but know specific parts of the URL.

For instance, if you are building a network browsing panel that lets users choose the protocol, host name, port number, and filename, you can construct a URL from those components. The following snippet demonstrates creating a URL for a page named **page1.html** on the example.com site:

```
new URL("http", "example.com", "/pages/page1.html");
```

This is equivalent to:

```
new URL("http://example.com/pages/page1.html");
```

The first argument specifies the protocol.

The second specifies the host name.

The last specifies the path to the file, starting with a / to indicate it is relative to the root of the host.

You can also include a port number with the following constructor:

URL gamelan = new URL("http", "example.com", 80, "pages/page1.html");

This creates a URL object for:

http://example.com:80/pages/page1.html

If you construct a URL object using one of these methods, you can retrieve the complete URL address as a string using the **toString** method or the equivalent **toExternalForm** method.

All four URL constructors throw a **MalformedURLException** if the provided arguments are invalid, such as specifying a null or unknown protocol. To handle this exception, you can use a try/catch block like this:

```
try {
   URL myURL = new URL(. . .)
} catch (MalformedURLException e) {
   // Exception handling code
}
```

Note that URL objects are immutable, meaning that once a URL object is created, its components (protocol, host name, filename, port number) cannot be changed.

Parsing a URL

The URL class provides several methods to retrieve information from a URL object. These methods allow you to access the protocol, authority, host name, port number, path, query, filename, and reference components:

- **getProtocol**: Returns the protocol component of the URL.
- **getAuthority**: Returns the authority component.
- **getHost**: Returns the host name.
- **getPort:** Returns the port number (returns -1 if not set).
- **getPath**: Returns the path component.

- **getQuery**: Returns the query string.
- **getFile**: Returns the filename, including the query string (if any).
- **getRef**: Returns the reference (fragment) of the URL.

Note that not all URLs contain every component. These methods are primarily designed for use with HTTP URLs, which usually have these components, making the URL class somewhat HTTP-centric.

Regardless of which constructor you use to create the URL object, these accessor methods allow you to query the URL's components.

Here is an example program that creates a URL object from a string and then uses the accessor methods to parse and print different components of the URL:

```
1. import java.net.*;
2. import java.io.*;
3.
4. public class ParseURL {
5.    public static void main(String[] args) throws Exception {
6.       URL aURL = new URL("http://example.com:80/docs/
   books/tutorial"
7.                          + "/index.html?
   name=networking#DOWNLOADING");
8.       System.out.println("protocol = " + aURL.getProtocol());
9.       System.out.println("authority = " + aURL.getAuthority());
10.      System.out.println("host = " + aURL.getHost());
11.      System.out.println("port = " + aURL.getPort());
12.      System.out.println("path = " + aURL.getPath());
13.      System.out.println("query = " + aURL.getQuery());
14.      System.out.println("filename = " + aURL.getFile());
15.      System.out.println("ref = " + aURL.getRef());
16.   }
17. }
```

The program will give the following output:

```
protocol = http
authority = example.com:80
host = example.com
port = 80
path = /docs/books/tutorial/index.html
query = name=networking
```

filename = /docs/books/tutorial/index.html?name=networking
ref = DOWNLOADING

This example demonstrates how to extract various components of a URL using the URL class, saving you the need to manually parse URLs.

Reading directly from a URL

Once a URL object is created, you can use its **openStream()** method to retrieve a stream for reading the contents of the URL. The **openStream()** method returns a **java.io.InputStream** object, making reading from a URL as straightforward as reading from an input stream.

The following Java program demonstrates how to use **openStream()** to obtain an input stream for the URL http://www.oracle.com/. It then wraps the input stream in a **BufferedReader**, which is used to read the content of the URL line by line. The content read is printed to the standard output:

```
1. import java.net.*;
2. import java.io.*;
3.
4. public class URLReader {
5.     public static void main(String[] args) throws Exception {
6.         URL oracle = new URL("http://www.oracle.com/");
7.         BufferedReader in = new BufferedReader(
8.                 new InputStreamReader(oracle.openStream()));
9.
10.        String inputLine;
11.
12.        while ((inputLine = in.readLine()) != null)
13.            System.out.println(inputLine);
14.
15.        in.close();
16.    }
17. }
```

When you run this program, you should see the HTML code and textual content of the page located at http://www.oracle.com/ scrolling in the console. However, if the program hangs or throws an exception, you might need to configure a proxy host to enable

the program to connect to the Oracle server.

URL.openStream() is a quick and simple way to read content from a web resource. It returns an **InputStream** directly from the URL, making it easy to access the content of the web page. It can be used when you only need to read from a URL and do not need to configure the connection or interact with the resource in a complex way. It does not provide any ability to set request headers (like user-agent or cookies) or any control over connection timeouts. It does not even support writing data (e.g., in a POST request).

URLConnection

URLConnection is a more versatile and configurable class. It allows both reading from and writing to a resource and provides control over various aspects of the connection. Use URLConnection when you need more control, such as setting request headers, handling response codes, or sending data (e.g., making POST requests).

Here is an example of using URLConnection to get HTTP response headers while reading from URL:

```
1. import java.net.*;
2. import java.io.*;
3.
4. public class URLConnectionExample {
5.     public static void main(String[] args) {
6.         try {
7.             URL url = new URL("http://www.example.com");
8.             URLConnection connection = url.openConnection();
9.
10.            // Set request headers
11.            connection.setRequestProperty("User-Agent", "MyApp");
12.
13.            // Connect to the URL
14.            connection.connect();
15.
16.            // Read response headers
17.            System.out.println("Content-
    Type: " + connection.getHeaderField("Content-Type"));
18.            System.out.println("Date: " + connection.getHeaderField("Date'
19.            System.out.println("Server: " + connection.getHeaderField("Ser
20.
```

```
            // Read content from the URL
            BufferedReader in = new BufferedReader(new InputStreamRea
            String inputLine;
            while ((inputLine = in.readLine()) != null) {
                System.out.println(inputLine);
            }
            in.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

In the above program, the **openConnection()** method is called on the URL object to create a URLConnection object. This object allows for more advanced communication with the URL.

Unlike **openStream()**, URLConnection provides control over the request and access to additional information, such as headers and response metadata. The **setRequestProperty()** method is used to set a custom request header. In this case, the **User-Agent** header is set to **MyApp**. The **User-Agent** is typically used by web browsers or clients to identify themselves to the server. The **connect()** method establishes the actual connection to the server. While this is not always necessary (because calling **getInputStream()** would implicitly connect), it is a good practice to explicitly call **connect()** when setting headers or adjusting connection parameters. The **getHeaderField()** method is used to retrieve specific response headers from the server. This example retrieves the **Content-Type** (MIME type of the response), **Date** (the time when the response was generated), and **Server** (information about the server handling the request). These headers provide useful metadata about the resource being accessed. The **getInputStream()** method is called to obtain an **InputStream** from the URL, which allows reading the content. The **InputStreamReader** is wrapped around the **InputStream** to convert bytes into characters, and then a **BufferedReader** is used to read lines of text efficiently.

InetAddress class

The InetAddress class in Java is part of the java.net package and is

used to represent an IP address, both IPv4 and IPv6. It allows Java programs to resolve hostnames to IP addresses and vice versa, enabling communication over the internet or a local network.

The InetAddress class abstracts the concept of both IP addresses and DNS lookup, making it easier to work with network addresses. It can also represent a loopback address, a local address, or a broadcast address. InetAddress Objects can be of different types:

- **Host address**: Represents the IP address of a specific host (e.g., 192.168.1.10).
- **Loopback address**: The loopback address (e.g., 127.0.0.1) refers to the local machine.
- **Wildcard address**: Represents any IP address (useful for listening to incoming connections).
- **Broadcast address**: Sends data to all machines on a network.

The InetAddress class contains several useful methods for network programming. The following methods are used for getting the IP address and hostname:

- **getHostAddress()**: Returns the IP address as a String.
- **getHostName():** Returns the hostname of the system.
- **getCanonicalHostName()**: Returns the **fully qualified domain name** (**FQDN**) for the IP address.

For resolving hostnames to IP addresses, the methods are as follows:

- **getByName(String host)**: Resolves the hostname to an IP address, returning an InetAddress object.
- **getAllByName(String host)**: Resolves the hostname and returns an array of InetAddress objects, since a hostname may map to multiple IP addresses (especially in cases of load balancing).

The following methods are used for creating an InetAddress Object:

- **getByAddress(byte[] addr)**: Returns an InetAddress object based on a byte array representing the IP address.
- **getLocalHost()**: Returns the InetAddress object that represents the local machine.

Some of the methods available for checking special addresses:

- **isAnyLocalAddress()**: Checks if the address is a wildcard address (0.0.0.0).
- **isLoopbackAddress()**: Checks if the address is a loopback address (127.0.0.1 or ::1).
- **isMulticastAddress()**: Checks if the address is a multicast address.
- **isReachable(int timeout):** Tests if the address is reachable within a given timeout period.

Let us explore a few common operations using the InetAddress class.

Resolving a Hostname to an IP Address:

```
1. import java.net.*;
2.
3. public class InetAddressExample {
4.     public static void main(String[] args) {
5.         try {
6.             InetAddress address = InetAddress.getByName("www.google.co
7.             System.out.println("Host Name: " + address.getHostName());
8.             System.out.println("IP Address: " + address.getHostAddress());
9.         } catch (UnknownHostException e) {
10.            e.printStackTrace();
11.        }
12.    }
13. }
```

The **getByName()** method resolves the Hostname www.google.com to its corresponding IP address. The program prints the host name and IP address to the console.

The output will be as follows:

Host Name: www.google.com
IP Address: 142.250.185.132

Getting the Local Host IP Address:

```
1. import java.net.*;
2.
3. public class LocalHostExample {
4.     public static void main(String[] args) {
```

```
        try {
            InetAddress localHost = InetAddress.getLocalHost();
            System.out.println("Local Host Name: " + localHost.getHostNam
            System.out.println("Local IP Address: " + localHost.getHostAdd
        } catch (UnknownHostException e) {
            e.printStackTrace();
        }
    }
}
```

The **getLocalHost()** method is used to retrieve the local machine's IP address. It returns the loopback address if no IP address is configured.

Checking if an address is reachable:

```
import java.net.*;

public class ReachabilityTest {
    public static void main(String[] args) {
        try {
            InetAddress address = InetAddress.getByName("www.oracle.co
            boolean reachable = address.isReachable(5000); // Timeout i
            System.out.println("Is www.oracle.com reachable? " + reachabl
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

The **isReachable()** method checks whether the specified address is reachable within a given timeout (in this case, 5 seconds).

Resolving multiple IP addresses for a Hostname:

```
import java.net.*;

public class MultipleAddressExample {
    public static void main(String[] args) {
        try {
            InetAddress[] addresses = InetAddress.getAllByName("www.m
            for (InetAddress address : addresses) {
                System.out.println(address);
            }
```

```
        } catch (UnknownHostException e) {
            e.printStackTrace();
        }
    }
}
```

The **getAllByName()** method is used to return all IP addresses associated with a hostname. This is useful when a website has multiple IP addresses (due to load balancing, etc.).

TCP/IP client sockets and server sockets

In Java, the Socket and ServerSocket classes from the **java.net** package provide a way to establish communication between two devices over a network using the TCP/IP. TCP is a reliable, connection-oriented protocol that ensures data is transmitted accurately and in the correct order between a client and a server. In a TCP/IP network communication, the server waits for incoming connection requests from clients. The client initiates a connection to the server. Once the connection is established, the two systems can send and receive data in a reliable stream.

ServerSocket

The ServerSocket class in Java is part of the java.net package and is used to create a server that listens for incoming client connections. It plays a vital role in the server side of a client-server architecture, specifically for handling TCP-based communication. The ServerSocket is responsible for waiting for connection requests, establishing a connection, and handing off the communication to a Socket object. Once connected, the server and client can exchange data over the established connection. The ServerSocket binds to a specific port on the server, waiting for connection requests from clients. Once a connection is established, the ServerSocket can create separate Socket instances for each client, allowing multiple clients to connect to the server simultaneously. The **accept()** method of ServerSocket is a blocking call, meaning it waits until a client connects before proceeding. It uses TCP, which ensures reliable, ordered, and error-checked data transmission between client and server.

The following are some constructors provided by ServerSocket class:

- **ServerSocket(int port)**: Creates a server socket that listens on the specified port.

  The port is essential because clients will connect to this port to initiate communication.

  Example:

  ServerSocket serverSocket = new ServerSocket(8080); // Listens on port 8080

  ServerSocket(int port, int backlog):

  Similar to the previous constructor, but allows you to specify the backlog parameter, which indicates the maximum number of clients that can wait in the queue while the server is busy.

- **ServerSocket(int port, int backlog, InetAddress bindAddr)**: This constructor allows you to specify the IP address to bind to (for example, to bind to a specific network interface).

Important methods in ServerSocket

The following are the methods in ServerSocket:

- **accept()**: Waits for an incoming client connection and returns a Socket object for communication with the client. This method blocks until a client connects. It is written as follows:

  Socket clientSocket = serverSocket.accept(); // Waits for a client to connect

- **close()**: Closes the ServerSocket, stopping it from accepting new connections:

  serverSocket.close(); // Closes the server socket

- **getLocalPort()**: Returns the port that the server is listening on.

- **setSoTimeout(int timeout)**: Sets a timeout for the **accept()** method. If no connection is made within the specified time, the **accept()** method throws a **SocketTimeoutException**.

- **isBound()**: Checks if the server socket is bound to an

address.

- **isClosed()**: Checks if the server socket is closed.

Let us see an example demonstrating a simple server that listens on port 8080, accepts client connections, and communicates with the client by sending and receiving messages:

```
1. import java.io.*;
2. import java.net.*;
3.
4. public class SimpleServer {
5.     public static void main(String[] args) {
6.         try {
7.             // Create a server socket that listens on port 8080
8.             ServerSocket serverSocket = new ServerSocket(8080);
9.             System.out.println("Server is listening on port 8080");
10.
11.            // Accept a client connection
12.            Socket clientSocket = serverSocket.accept();
13.            System.out.println("Client connected");
14.
15.            // Get input and output streams for communication
16.            BufferedReader in = new BufferedReader(new InputStreamRea
17.            PrintWriter out = new PrintWriter(clientSocket.getOutputStrea
18.
19.            // Read a message from the client and send a response
20.            String clientMessage = in.readLine();
21.            System.out.println("Received from client: " + clientMessage);
22.            out.println("Hello, client! You sent: " + clientMessage);
23.
24.            // Close the client socket
25.            clientSocket.close();
26.            serverSocket.close();
27.        } catch (IOException e) {
28.            e.printStackTrace();
29.        }
30.    }
31. }
```

The server socket is created to listen on port 8080. The server blocks the **accept()** method until a client connects. Once a client is

connected, a Socket object (**clientSocket**) is returned for communication with the client. The server reads the client's message using **BufferedReader** and responds with **PrintWriter.** The server sends an acknowledgment message back to the client. Both the **clientSocket** and the s**erverSocket** are closed after the communication ends.

Handling multiple clients

To handle multiple clients simultaneously, the server needs to handle each client connection in a separate thread. Here is how it works. The server continues listening for connections while creating a new thread to handle each connected client. Each client is processed independently by a separate thread, ensuring that the server can handle multiple connections concurrently.

Let us see an example of a multithreaded server:

```
1. import java.io.*;
2. import java.net.*;
3.
4. public class MultiThreadedServer {
5.     public static void main(String[] args) {
6.         try {
7.             ServerSocket serverSocket = new ServerSocket(8080);
8.             System.out.println("Server is listening on port 8080");
9.
10.            while (true) {
11.                // Accept a client connection
12.                Socket clientSocket = serverSocket.accept();
13.                System.out.println("Client connected");
14.
15.                // Handle each client in a new thread
16.                new ClientHandler(clientSocket).start();
17.            }
18.        } catch (IOException e) {
19.            e.printStackTrace();
20.        }
21.    }
22. }
23.
24. class ClientHandler extends Thread {
```

```
25.     private Socket clientSocket;
26. 
27.     public ClientHandler(Socket socket) {
28.         this.clientSocket = socket;
29.     }
30. 
31.     public void run() {
32.         try {
33.             BufferedReader in = new BufferedReader(new InputStreamRea
34.             PrintWriter out = new PrintWriter(clientSocket.getOutputStrea
35. 
36.             // Communicate with the client
37.             String clientMessage = in.readLine();
38.             System.out.println("Received from client: " + clientMessage);
39.             out.println("Hello, client! You sent: " + clientMessage);
40. 
41.             // Close the client socket
42.             clientSocket.close();
43.         } catch (IOException e) {
44.             e.printStackTrace();
45.         }
46.     }
47. }
```

The **MultiThreadedServer** listens for connections in an infinite loop. The while (true) loop keeps the server running indefinitely. It continuously waits for client connections using **serverSocket.accept()**. When a connection is accepted, it creates a new **ClientHandler** thread to handle the client and starts it. This allows the server to handle multiple clients at the same time.

Timeouts period of ServerSocket can be handled by setting a timeout for the **accept()** method, which will allow the server to stop waiting for client connections after a certain period. This is useful when you do not want the server to wait indefinitely. The following code shows setting a timeout period of 10 seconds.

serverSocket.setSoTimeout(10000); // Set timeout to 10 seconds

If no client connects within 10 seconds, a **SocketTimeoutException** is thrown.

Some common exceptions in ServerSocket are listed below:

- **IOException**: This is a general exception thrown if an I/O error occurs when opening, accepting, or communicating with a socket.
- **BindException**: Thrown if the server attempts to bind to a port that is already in use.
- **SocketTimeoutException**: Thrown if the **accept()** method times out while waiting for a connection (when a timeout is set using **setSoTimeout()**).

Socket class

The Socket class is part of the java.net package. It represents a client socket that connects to a server. The Socket class provides a way to create a client-side socket that connects to a server for sending and receiving data. Typically used with TCP, which ensures reliable, ordered, and error-checked delivery of data between applications. Some of its key Methods and Fields are listed below:

Constructors

Socket(String host, int port) throws UnknownHostException, IOException

Creates a socket and connects it to the specified port number at the specified IP address or hostname. **Host** indicates the hostname or IP address of the server and **port** is the port number on the server to which you want to connect.

For example:

```
Socket socket = new Socket("localhost", 1234);
```

Socket(InetAddress address, int port)

Creates a socket with a given IP address and port number.

For example:

```
InetAddress address = InetAddress.getByName("localhost");
Socket socket = new Socket(address, 1234);
```

Socket(String host, int port, InetAddress localAddr, int localPort)

Creates a socket and connects it to the specified remote port, using the specified local address and port for the connection.

For example:

```
Socket socket = new Socket("localhost", 1234,
```

InetAddress.getByName("192.168.1.1"), 0);

The frequently used methods of socket class are listed below:

	Description	
[illegible]	[illegible]	
[illegible]	[illegible]	
[illegible]	[illegible]	
[illegible]	[illegible]	
[illegible]	[illegible]	
[illegible]	[illegible]	
[illegible]	[illegible]	
[illegible]	[illegible]	
[illegible]	[illegible]	

***Table 11.1**: Commonly used methods of Socket class*

To write a client for the **SimpleServer** class, you will need a Java program that connects to the server's port (8080), sends a message to the server, and then receives and prints the response from the server.

```
1. import java.io.*;
2. import java.net.*;
3.
4. public class SimpleClient {
5.     public static void main(String[] args) {
6.         String host = "localhost";
7.         int port = 1234;
8.
9.         try (Socket socket = new Socket(host, port)) {
10.            // Create input and output streams
11.            PrintWriter out = new PrintWriter(socket.getOutputStream(), t
12.            BufferedReader in = new BufferedReader(new InputStreamRea
13.
14.            // Send a message to the server
15.            out.println("Hello, Server!");
16.
17.            // Read the server's response
18.            String response = in.readLine();
19.            System.out.println("Server says: " + response);
20.
21.        } catch (IOException e) {
```

```
            e.printStackTrace();
        }
    }
}
```

There should be proper error handling in the code. The most common exception you will encounter with sockets is **IOException**. It is thrown for various I/O errors, such as a failure to connect or an error during communication. **SocketTimeoutException** exception is thrown if a read operation times out.

Datagram communication with UDP

Using UDP in Java involves working with the DatagramSocket and DatagramPacket classes, which are part of the **java.net** package. UDP is a connectionless protocol that provides faster, but less reliable, communication compared to TCP. It does not guarantee the delivery, order, or integrity of packets, but it is useful for scenarios where speed is crucial, such as in real-time applications like video streaming or online gaming.

DataGramSocket class

The DatagramSocket class is used for implementing UDP network communication. Unlike TCP sockets, which provide a connection-oriented communication channel, UDP is connectionless and provides a way to send datagrams (packets of data) between hosts. The DatagramSocket class is fundamental for developing applications that require fast, lightweight, and connectionless communication.

Constructors of DatagramSocket:

	Explanation	
DatagramSocket()	DatagramSocket socket = new DatagramSocket();	port on the local machine.
DatagramSocket(int port)	new DatagramSocket(9876);	machine.
DatagramSocket(int port, InetAddress address)	new DatagramSocket(9876, InetAddress.getByName("192.168.1.10"));	(9876) and IP address (192.168.1.10).

***Table 11.2**: Commonly used constructors of DatagramSocket class*

The following are the important methods of DatagramSocket:

	Explanation	

void send(DatagramPacket p)	Sends a datagram packet through the socket. This method is used to send UDP packers over the network.
void receive(DatagramPacket p)	Receives a datagram packet from this socket. This method blocks until a packet is received.
void close()	Closes the socket, freeing up any resources bound to it.
void bind(SocketAddress addr)	Binds the socket to a specific address and port.
InetAddress getInetAddress()	Returns the address to which this socket is connected (only relevant for connected sockets).
InetAddress getLocalAddress()	Returns the local address to which this socket is bound.
int getPort()	Returns the port number of the remote host connected to this socket (only relevant for connected sockets).
int getLocalPort()	Returns the local port number to which this socket is bound on the local machine.
void connect(InetAddress address, int port)	Connects the socket to a specific address and port, meaning only packers from this destination will be accepted.
void disconnect()	Disconnects the socket from the remote address and port, allowing it to receive packers from any address.
boolean isBound()	Returns true if the socket is bound to an IP address and port number.
boolean isClosed()	Returns true if the socket is closed.
int getSoTimeout()	Returns the timeout value in milliseconds for blocking receive operations.
void setSoTimeout(int timeout)	Sets the timeout value in milliseconds (5 seconds) for blocking receive operations. If no packet is received within the timeout period, a SocketTimeoutException is thrown.
int getSendBufferSize()	Returns the size of the send buffer, which determines how much data can be queued for sending.
void setSendBufferSize(8192)	Sets the send buffer to 8192 bytes. This controls how much data can be stored in the buffer before it's sent out.
int getReceiveBufferSize()	Returns the size of the receive buffer, which determines how much data can be queued for receiving.
void setReceiveBufferSize(8192)	Sets the receive buffer to 8192 bytes. This controls how much data can be stored in the buffer before being processed.

***Table 11.3**: Commonly used methods of DatagramSocket class*

Sending data:

- **send(DatagramPacket p)**: Sends a datagram packet from this socket.

  DatagramPacket packet = new DatagramPacket(data, data.length, InetAddress.getByName("localhost"), 1234);
  socket.send(packet);

  The DatagramPacket object contains the data to be sent, the length of the data, and the destination address and port.

Receiving data:

- **receive(DatagramPacket p)**: Receives a datagram packet

from this socket.

DatagramPacket packet = new DatagramPacket(buffer, buffer.length);

socket.receive(packet);

The DatagramPacket object is used to store the received data.

DatagramPacket class

The DatagramPacket class in Java, represents a packet of data that can be sent or received over a network using the UDP. It encapsulates the data being sent or received, along with the address and port information required to deliver the data to its destination. DatagramPacket encapsulates data in the form of a byte array, along with destination information (address and port) for sending or receiving. It provides a simple way to handle data transmission in a connectionless protocol like UDP.

DatagramPacket class constructors and methods with examples:

	Explanation	
DatagramPacket(byte[] buf, int length)	Creates a packet for receiving data with a buffer of specified length.	**DatagramPacket packet = new DatagramPacket(buffer, 1024);**
DatagramPacket(byte[] buf, int length, InetAddress address, int port)	Creates a packet for sending data to a specified address and port.	**DatagramPacket packet = new DatagramPacket(message, message.length, InetAddress.getByName("example.com"), 9876);**
DatagramPacket(byte[] buf, int offset, int length)	Creates a packet for receiving data with a buffer, offset, and length.	**DatagramPacket packet = new DatagramPacket(buffer, 5, 10);** The data will be placed starting from the 5th position in the buffer.
DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)	Creates a packet for sending data with a buffer, offset, length, destination address, and port.	**DatagramPacket packet = new DatagramPacket(message, 5, 10, InetAddress.getByName("example.com"), 9876);**

***Table 11.4**: Commonly used constructors of DatagramPacket class*

DatagramPacket is required to send and receive data. For sending and receiving data packets how datagramPacket is created is shown below:

For sending data:

DatagramPacket(byte[] buf, int length, InetAddress address, int port)

- **buf**: The byte array containing the data to be sent.
- **length**: The length of the data to be sent.
- **address**: The destination IP address.
- **port**: The destination port number.

Example:

```
byte[] data = "Hello, Server!".getBytes();
InetAddress address = InetAddress.getByName("localhost");
int port = 9876;
DatagramPacket packet = new DatagramPacket(data, data.length,
address, port);
```

For receiving data:

DatagramPacket(byte [] buf, int length)

- **buf**: The byte array to store incoming data.
- **length**: The length of the buffer.

Example:

```
byte[] buffer = new byte[1024];
DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
```

Commonly used methods of DatagramPacket class are listed below:

	Explanation	
byte[] getData()	Returns the data in the packet.	
void setData(byte[] buf)	Sets the data buffer for the packet. Useful when you need to change the content or size of the buffer for reuse.	
int getLength()	Returns the length of the data stored in the packet.	
void setLength(int length)	Sets the length of the data to be sent or received.	
InetAddress getAddress()	Returns the IP address associated with the packet.	
int getPort()	Returns the port number of the remote host to which the packet was sent or from which the packet was received.	
void setAddress(InetAddress address)	Sets the IP address for the packet when sending data.	
void setPort(int port)	Sets the port number for the packet when sending data.	

***Table 11.5**: Commonly used methods of DatagramPacket class*

This example demonstrates how to implement a simple UDP client-server application where the client sends a message, and the server responds to it:

Server code (UDP server):

```
1. import java.net.DatagramPacket;
2. import java.net.DatagramSocket;
3. import java.net.InetAddress;
4.
5. public class UDPServer {
6.     public static void main(String[] args) {
```

```
7.      try {
8.          // Create a DatagramSocket to listen on port 9876
9.          DatagramSocket serverSocket = new DatagramSocket(9876);
10.
11.         // Buffer to receive incoming data
12.         byte[] receiveBuffer = new byte[1024];
13.
14.         System.out.println("Server is waiting for client messages...");
15.
16.         // Infinite loop to keep the server running
17.         while (true) {
18.             // Create a packet to receive data
19.             DatagramPacket receivePacket = new DatagramPacket(recei
20.             // Receive the data
21.             serverSocket.receive(receivePacket);
22.
23.             // Extract the client's message and IP address
24.             String clientMessage = new String(receivePacket.getData(),
25.             InetAddress clientAddress = receivePacket.getAddress();
26.             int clientPort = receivePacket.getPort();
27.
28.             System.out.println("Received message from client: " + client
29.
30.             // Prepare the response message
31.             String responseMessage = "Hello from server!";
32.             byte[] responseBuffer = responseMessage.getBytes();
33.
34.             // Send the response to the client
35.             DatagramPacket responsePacket = new DatagramPacket(resp
36.             serverSocket.send(responsePacket);
37.
38.             System.out.println("Response sent to client: " + responseMes
39.         }
40.     } catch (Exception e) {
41.         e.printStackTrace();
42.     }
43.   }
44. }
```

Client code (UDP client):

```java
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class UDPClient {
    public static void main(String[] args) {
        try {
            // Create a DatagramSocket to send data
            DatagramSocket clientSocket = new DatagramSocket();

            // Server address and port
            InetAddress serverAddress = InetAddress.getByName("localhos
            int serverPort = 9876;

            // Client's message
            String message = "Hello from client!";
            byte[] sendBuffer = message.getBytes();

            // Create a packet to send data to the server
            DatagramPacket sendPacket = new DatagramPacket(sendBuffe

            // Send the packet to the server
            clientSocket.send(sendPacket);

            System.out.println("Message sent to server: " + message);

            // Buffer to receive the response from the server
            byte[] receiveBuffer = new byte[1024];

            // Create a packet to receive the response
            DatagramPacket receivePacket = new DatagramPacket(receive

            // Receive the response from the server
            clientSocket.receive(receivePacket);

            // Extract the server's response message
            String serverMessage = new String(receivePacket.getData(), 0,
            System.out.println("Response from server: " + serverMessage);

```

```
            // Close the socket
            clientSocket.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

In UDP Server, **DatagramSocket serverSocket = new DatagramSocket(9876);** creates a **DatagramSocket** bound to port 9876 to listen for incoming UDP packets. server**Socket.receive(receivePacket);**. The server waits to receive a datagram packet from the client. **InetAddress clientAddress = receivePacket.getAddress();**. Extracts the client's IP address. **serverSocket.send(responsePacket);**. Sends a response back to the client using the extracted IP address and port.

In UDP Client, **DatagramSocket clientSocket = new DatagramSocket();** creates a DatagramSocket that will send the packet to the server. **clientSocket.send(sendPacket);**: The client sends a packet to the server containing its message. **clientSocket.receive(receivePacket);**: The client waits for the server's response.

How the program works:

1. The client sends a **Hello from client!** message to the server.
2. The server receives this message and sends back a **Hello from server!** response.
3. The client receives this response and prints it to the console.

Example programs

Below is a simple Java client program that connects to an HTTP server, sends a GET request, and retrieves the HTML content of a webpage, printing the response to the console:

```
import java.io.*;
import java.net.*;

public class HttpClient {
    public static void main(String[] args) {
```

```
6.      String host = "www.example.com"; // Change this to the host you
7.      int port = 80; // HTTP default port
8.
9.      try (Socket socket = new Socket(host, port)) {
10.         // Send HTTP GET request
11.         PrintWriter out = new PrintWriter(socket.getOutputStream(), true)
12.         out.println("GET / HTTP/1.1");
13.         out.println("Host: " + host);
14.         out.println("User-Agent: JavaHttpClient");
15.         out.println("Connection: close");
16.         out.println(); // Blank line to indicate end of headers
17.
18.         // Read response from the server
19.         BufferedReader in = new BufferedReader(new InputStreamReader(
20.         String inputLine;
21.         while ((inputLine = in.readLine()) != null) {
22.             System.out.println(inputLine);
23.         }
24.
25.     } catch (IOException e) {
26.         e.printStackTrace();
27.     }
28.   }
29. }
```

A socket is created using **Socket(host, port)** to connect to the server on the specified host and port (default for HTTP is 80). The HTTP GET request is manually sent using a **PrintWriter**. This includes the necessary headers such as Host, User-Agent, and Connection: close to signal the server to close the connection after the response.

The response from the server is read using a **BufferedReader** that reads from the socket's input stream. The HTML content of the requested page is printed line by line. The try-with-resources block ensures that the socket and I/O streams are closed after use.

The program will print the raw HTML content of the webpage to the console.

Note: Replace www.example.com with the actual URL or domain

you want to fetch data from.

Let us see another example of a simple Java TCP-based port scanner that checks a range of ports on a remote machine to see which ones are open. This program attempts to connect to each port in the specified range and reports which ports are open.

```
1. import java.net.Socket;
2.
3. public class PortScanner {
4.     public static void main(String[] args) {
5.         String targetHost = "127.0.0.1"; // The IP address or hostname o
6.         int startPort = 1; // Starting port number
7.         int endPort = 1024; // Ending port number
8.
9.         System.out.println("Scanning ports on host: " + targetHost);
10.
11.        for (int port = startPort; port <= endPort; port++) {
12.            try {
13.                // Try to establish a socket connection to the given host and p
14.                Socket socket = new Socket(targetHost, port);
15.                System.out.println("Port " + port + " is open.");
16.                socket.close(); // Close the socket after successful connection
17.            } catch (Exception e) {
18.                // If connection fails, the port is closed or unreachable
19.                // Optionally print the closed ports by uncommenting the next
20.                // System.out.println("Port " + port + " is closed.");
21.            }
22.        }
23.
24.        System.out.println("Port scanning complete.");
25.    }
26. }
```

The program scans ports from **startPort** to **endPort** on a target host (in this case, 127.0.0.1, which is localhost). For each port in the range, the program tries to establish a Socket connection to the target host. If a connection is successful, it means the port is open. If a connection fails (e.g., due to **IOException** or **SocketTimeoutException**), it means the port is either closed or filtered.

The program prints the port numbers that are open.

For running the program replace 127.0.0.1 with the IP address or hostname of the machine you want to scan. You can adjust the **startPort** and **endPort** values to scan a specific range of ports.

Example output:

Scanning ports on host: 127.0.0.1
Port 22 is open.
Port 80 is open.
Port 443 is open.
Port scanning complete.

Note that port scanning can be intrusive. Make sure you have permission to scan the target machine. Scanning many ports in a short time could trigger security systems, so be cautious when scanning external networks.

Conclusion

In this chapter, we explored the core concepts of networking in Java. We began by introducing the basics of networking and understanding how communication over networks is achieved through sockets. The Socket API in Java provides the foundation for both client-server communication and peer-to-peer connections, offering key classes such as Socket, ServerSocket, and InetAddress. We also explored how Java interacts with web resources using the URL and URLConnection classes, enabling communication over HTTP.

Furthermore, we covered the InetAddress class, which simplifies the handling of IP addresses, and examined how TCP/IP client-server sockets facilitate reliable, connection-based communication. Finally, we discussed datagram communication via UDP, which provides a lightweight alternative for connectionless data transfer using DatagramSocket and DatagramPacket classes.

In the next chapter, we will explore Web Component Development with Servlets.

Exercises

1. Simple TCP client-server communication: Write a program where a server listens for incoming connections on a specific

port. The client should connect to the server and send a message like Hello, Server!. The server should respond with Hello, Client! and close the connection.

2. Multi-client server: Create a server that can handle multiple clients concurrently using threads. Each client should send a unique message to the server, and the server should respond with a greeting. Ensure that multiple clients can connect and receive a response without the server blocking.
3. UDP communication: Implement a simple UDP client-server program where the client sends a message like Hello, Server! to the server using a DatagramSocket. The server should receive the message and send a response like "Hello, Client!" back to the client.
4. File transfer using TCP: Write a TCP-based file transfer application where a client can request a file from the server. The server should send the requested file, and the client should save the file locally.
5. Echo server: Create a TCP echo server where clients can send a message, and the server will reply with the same message. The client should continue to send messages until it sends the message exit, which will close the connection.
6. Simple chat application: Implement a simple TCP-based chat system where multiple clients can connect to a server. Each client should be able to send messages to the server, which will broadcast the message to all connected clients.
7. Time server: Write a server that listens on a specific port and sends the current date and time to the client when it connects. The client should connect, display the received time, and then disconnect.
8. Client-server calculator: Create a client-server application where the client sends an arithmetic expression (like 5 + 3) to the server. The server should evaluate the expression and send the result back to the client.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the

Authors:

https://discord.bpbonline.com

Chapter 12

Web Component Development with Servlets

Introduction

This chapter explores the essential components of web development using servlets within the **Java 2 Enterprise Edition (J2EE)** framework. Servlets are pivotal in creating dynamic, robust, and scalable web applications by bridging client requests and server responses. The chapter begins by outlining the J2EE architecture, emphasizing its multi-tiered structure and its importance in enterprise-level applications. It explores the foundational principles of web applications, including the role of the **Hypertext Transfer Protocol (HTTP)** in client-server communication.

The discussion advances to web containers, highlighting their role in managing servlets and facilitating the servlet lifecycle. Detailed insights into the Servlet API, including core classes like HttpServlet, HttpServletRequest, and HttpServletResponse, provide a solid understanding of how servlets handle requests and generate

responses. Additionally, practical aspects of writing service methods, implementing state management, and leveraging ServletContext and ServletConfig for application-wide and servlet-specific configurations are examined. This chapter equips you with the foundational knowledge to design and develop interactive, efficient, and data-driven web applications.

Structure

The chapter covers the following topics:

- J2EE architecture
- Web application
- Hypertext Transfer Protocol
- Web containers
- Servlets
- Servlet life cycle
- Servlet API
- HttpServlet
- HttpServletRequest
- HttpServletResponse
- Writing service methods
- Handling multi-valued parameter
- State management
- ServletContext
- ServletConfig

Objectives

The objective of this chapter is to provide a comprehensive understanding of Servlets in the context of J**ava 2 Enterprise Edition (J2EE)**. By the end of this chapter, you will gain a clear grasp of the J2EE architecture and its role in building robust enterprise applications. You will learn how web containers manage servlets and will have a strong grasp of the servlet API. The chapter aims to make readers understand the Servlet life cycle and how

ServletContext and ServletConfig interfaces help in managing configurations and context in web applications. Additionally, you will explore state management techniques, both on the client and server side, enabling you to maintain user data across sessions and requests, which is essential for developing dynamic, data-driven web applications.

J2EE architecture

Java 2 Platform, Enterprise Edition (J2EE), now known as Jakarta EE, is a platform designed for developing multi-tiered, distributed, and web-based enterprise applications. It provides a robust, scalable, and simplified approach to enterprise application development, which includes core technologies like servlets, JSP, EJB, and web services. J2EE is built on top of the **Java Standard Edition (Java SE)** and extends its capabilities by offering APIs and a runtime environment for large-scale, distributed, transactional, and secure applications. It follows a multi-layered architecture that helps in separating concerns and promoting reusability, scalability, and maintainability. There are various architecture models in J2EE that define how these layers interact with each other.

Here are the primary architecture models in J2EE:

Single-tier architecture

This architecture refers to a system where the entire application resides on a single system. All components, such as the user interface, business logic, and data access, are housed in a single tier. In practice, this is rarely used in enterprise systems but serves as the foundation for understanding more complex architectures.

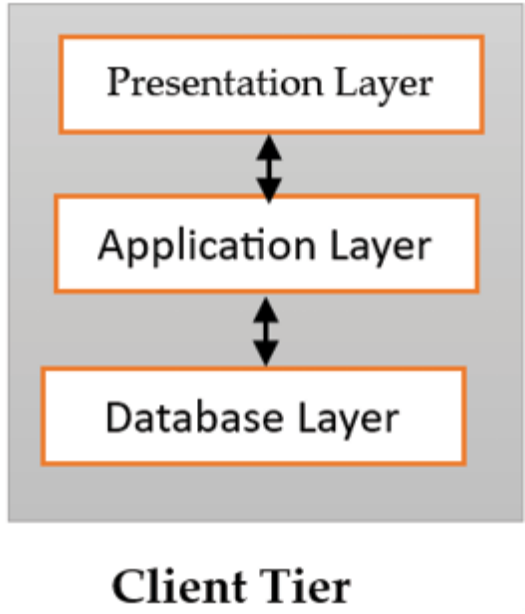


***Figure 12.1**: Single-tier architecture*

Two-tier architecture

In this model, the system is divided into two layers:

- **Client**: The client (typically a web browser) sends requests for resources such as HTML pages or data.
- **Server**: The server responds with the requested resources or data (could be a database server or a web server).

In this model, the web container (server) manages requests directly between the client and server. It is simple to implement but does not scale well for large applications. The following figure represents two tier architecture:

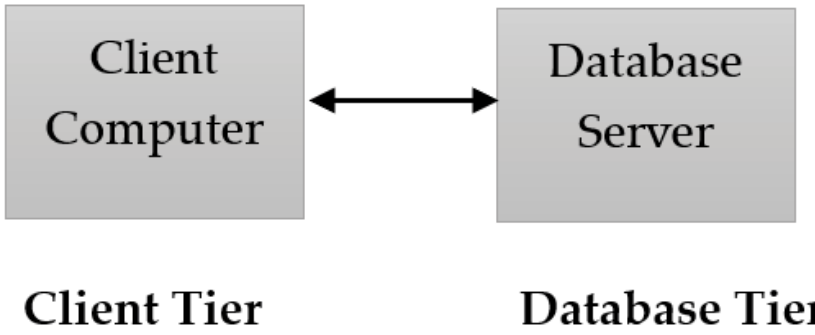


***Figure 12.2**: Two-tier architecture*

Three-tier architecture

This is a common architecture model for J2EE applications, which is divided into three logical layers:

- **Client tier (presentation layer)**: This layer represents the user interface, which could be an HTML page, JSP, or a client-side application like a browser. It interacts with the user and sends requests to the server or business logic tier.
- **Middle tier (business logic layer)**: This tier contains the core logic of the application. It is where servlets, **Enterprise JavaBeans (EJBs)**, or other business logic reside. The web container lies in this tier and processes HTTP requests from the client, invoking the appropriate servlets or JSPs, which in turn may interact with EJBs or other business components.
- **Data tier (database layer)**: This tier contains the database or any other data storage mechanism. It is responsible for storing, retrieving, and updating the data required by the application.

In the three-tier architecture, the web container sits in the middle tier and handles the communication between the client and business logic components. It ensures that business rules are applied correctly, and data is exchanged between the presentation and database layers. In this model, the business logic is separated from the user interface and the database, allowing for better scalability and maintainability. The following figure represents three tier

architecture:

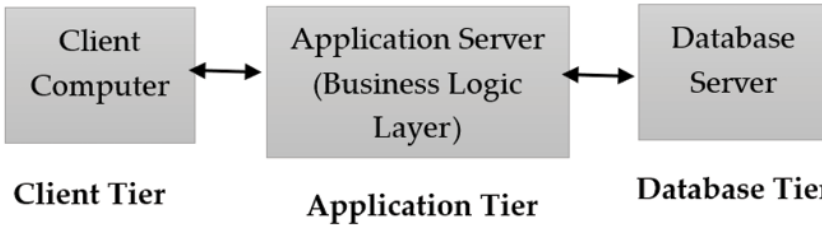


Figure 12.3: *Three-tier architecture*

N-tier architecture

An N-tier or multi-tier architecture is an extension of the three-tier model, where additional tiers can be added to further segregate the responsibilities of each layer. In N-tier architecture, the application is divided into more than three layers for better separation of concerns, scalability, and maintainability. Typically, an additional service layer or integration layer is introduced. This layer contains business services that interact with the database and external systems. The web container continues to manage communication between the client and middle-tier services, but the N-tier model allows for additional flexibility and modularity in complex enterprise applications. Common tiers include:

- **Client tier**: Includes web browsers or mobile applications that interact with the application.
- **Presentation tier**: A dedicated tier for handling the presentation logic, which could consist of web pages generated using **JavaServer Pages (JSP)** or **JavaServer Faces (JSF)**.
- **Business logic tier**: This tier handles the core business processing and can include **Enterprise JavaBeans** (**EJBs**), Servlets, or other business services.
- **Integration tier**: Handles interaction with external systems, such as web services or other enterprise applications through APIs, making use of technologies like JAX-RS or JAX-WS for RESTful and SOAP-based services, respectively.
- **Data access tier**: A dedicated layer for managing database

operations, abstracting the complexity of database connectivity from the other layers.

- **Data tier:** Contains the databases, files, or other storage mechanisms to persist data.

In this model, each tier is independent and can be scaled individually. The architecture becomes more complex but provides better modularity, scalability, and fault tolerance.

In J2EE Architecture, interaction begins with a client request. The client (web browser) sends an HTTP request to the server. The web container intercepts this request, finds the appropriate servlet or JSP to handle it, and passes the request and response objects to that component.

The servlet or JSP is responsible for business logic execution. The servlet or JSP can process the request, often by interacting with the EJBs or directly with the database (data tier). Once the processing is complete, the servlet/JSP generates a response, which is passed back to the client via the web container.

Web application

A web application is an application that runs on a web server, and users access it via a web browser or a specialized user agent over the internet or an intranet. The browser generates HTTP requests for specific URLs that correspond to resources on a web server. In response, the server processes these requests and sends back HTML pages for the browser to display. At the heart of a web application lies its server-side logic, which often operates within a structured architecture. A common example is the three-tier architecture, consisting of presentation, business, and data layers, each serving distinct functions within the application. Unlike traditional desktop applications, web applications provide dynamic interaction and real-time data processing, allowing users to perform a variety of tasks, such as submitting forms, managing content, or engaging in e-commerce, without needing to install software locally. They typically follow a client-server architecture, where the client interface is presented in the browser, and the server handles the business logic and data management. This model not only enhances accessibility across different devices and platforms but also

simplifies updates and maintenance, making web applications an essential component of modern software development. The following figure represents a typical web representation:

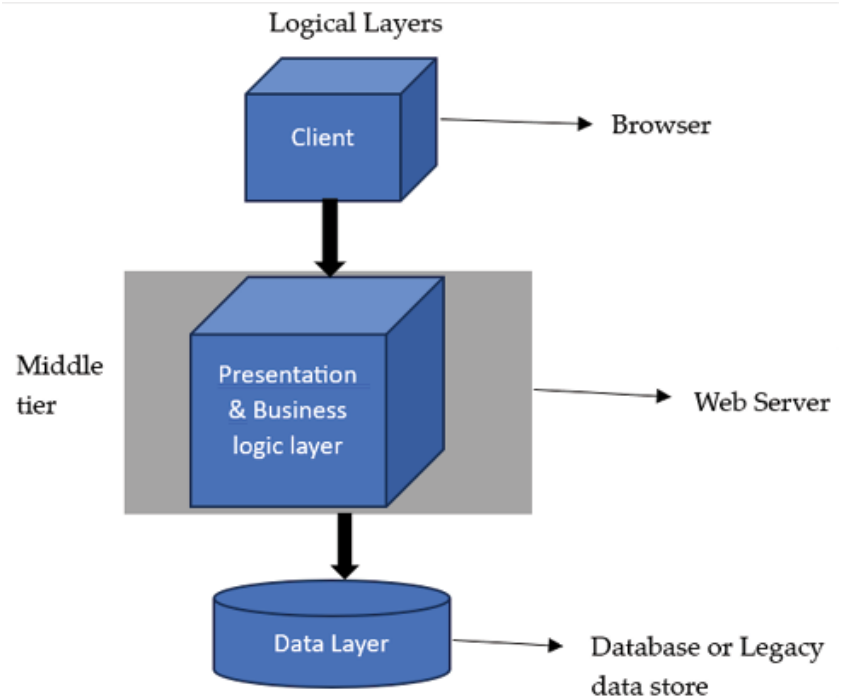


***Figure 12.4**: Typical web implementation*

Hypertext Transfer Protocol

The **Hypertext Transfer Protocol (HTTP)** is a protocol designed to facilitate communication between clients and servers. It operates as a request-response protocol, where a client (often a web browser) sends requests to a server hosting a website, which then responds with the appropriate data. HTTP protocol provides many request methods. Two primary HTTP request methods are:

- **GET**: This method requests data from a specified resource, such as retrieving a web page.
- **POST**: This method submits data to be processed to a specified resource, commonly used for form submissions.

In addition to GET and POST, there are several other HTTP request methods. Some of them are listed below:

Request	Description
HEAD	Similar to GET, but it returns only the HTTP headers without the document body.
PUT	This method uploads a representation of the specified resource (**Uniform Resource Identifier (URI)**).
DELETE	Deletes the specified resource from the server.
OPTIONS	This method returns the HTTP methods that the server supports for a specific resource.
CONNECT	It establishes a transparent TCP/IP tunnel to facilitate secure connections.

***Table 12.1**: HTTP request methods*

A web server is software that delivers web content, such as web pages, to clients (e.g., web browsers) via the HTTP protocol. It processes incoming HTTP requests and serves the appropriate responses.

Web containers

A **web container** (also known as a **servlet container**) is a part of a Java EE application server that manages the execution of J2EE web components such as servlets, **JavaServer Pages (JSPs)**, and filters. It provides the runtime environment for these web components to interact with clients, handle requests, and generate dynamic responses. A servlet container is responsible for managing the lifecycle of servlets, mapping URLs to specific servlets, and ensuring that the requester has the appropriate access rights. It plays a crucial role in executing server-side Java components like servlets and JSPs. The key responsibilities of web containers are listed below:

- **Lifecycle management:** The web container manages the life cycle of servlets, including their initialization, execution, and destruction. This simplifies the development process as the container takes care of threading, object pooling, and memory management.
- **Request handling**: It intercepts HTTP requests, invokes the appropriate servlet, and manages the request-response cycle.
- **Session management:** It manages session tracking (e.g.,

using cookies, URL rewriting, or hidden form fields), which is essential for maintaining client interactions.

- **Security:** Web containers enforce security constraints defined in the application's deployment descriptor (**web.xml**), such as access control for resources or enforcing authentication and authorization.
- **Java Naming and Directory Interface (JNDI) lookup services**: It provides JNDI services, allowing servlets and JSPs to look up resources like databases.
- Popular examples of web containers include Apache Tomcat, Jetty, and GlassFish.

Servlets

A Servlet is a Java-based web component that extends the capabilities of a server. Servlets are a key part of the Java **Enterprise Edition** (**EE**) platform and work within a web container to create web applications.

A servlet is a Java class used to process client requests (often HTTP requests) and generate dynamic responses (such as HTML, JSON, or XML). Servlets act as intermediaries between client requests (such as from a browser) and server responses, making it possible to create dynamic web applications using Java. The **javax.servlet** and **javax.servlet.http** packages provide interfaces and classes for writing servlets. All servlets must implement the servlet interface, which defines life-cycle methods. When implementing a generic service, you can extend the GenericServlet class provided with the Java Servlet API. To handle http request servlet should extend from HttpServlet class. The HttpServlet class provides methods, such as doGet and doPost, for handling HTTP-specific services. Some example use cases of servlets are:

- Processing form data from an HTML form
- Handling e-commerce transactions
- Serving dynamic web pages based on user input
- Interfacing with databases to retrieve and update data

Servlet life cycle

Servlets follow a specific lifecycle managed by the servlet container. The container handles the creation, initialization, request processing, and destruction of servlets. The servlet lifecycle consists of three main phases:

- **Initialization (init()):** The servlet is instantiated (i.e., an object of the servlet class is created) when it is first requested or when the web application starts. The container calls the init() method to initialize the servlet. This method is called only once during the lifecycle, typically to perform one-time startup processes such as resource allocation or database connection setup. **init ()** method is declared as follows:

  ```
  public void init(ServletConfig config) throws ServletException {
      // Initialization code here, executed only once
  }
  ```

- **Request handling (service())**: After initialization, the servlet remains in memory and can handle multiple client requests. Each time a request is made, the container calls the **service()** method. This method dispatches the request to the appropriate method (e.g., **doGet()** for GET requests or **doPost()** for POST requests). The **service()** method checks the HTTP request type and invokes the corresponding method (**doGet()**, **doPost()**, etc.). The **service ()** method is declared as follows:

  ```
  public void service(HttpServletRequest request,
  HttpServletResponse response) throws ServletException,
  IOException {
      // Request handling logic here
  }
  ```

Common HTTP methods handled by the **service()** method:

doGet(): Handles GET requests.

doPost(): Handles POST requests.

doPut(): Handles PUT requests.

doDelete(): Handles DELETE requests.

The signature of all these methods is similar to service method. The

declaration of doGet() and doPost() methods are shown below.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    // Handling GET request
}

protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    // Handling POST request
}
```

Destruction

The **destroy()** method is called once, just before the servlet is removed from memory by the container (usually when the server shuts down or the servlet is explicitly unloaded). This is where you should release any resources held by the servlet, such as database connections or file handles. The declaration of the destroy method is shown below:

```
public void destroy() {
    // Cleanup code here, executed only once
}
```

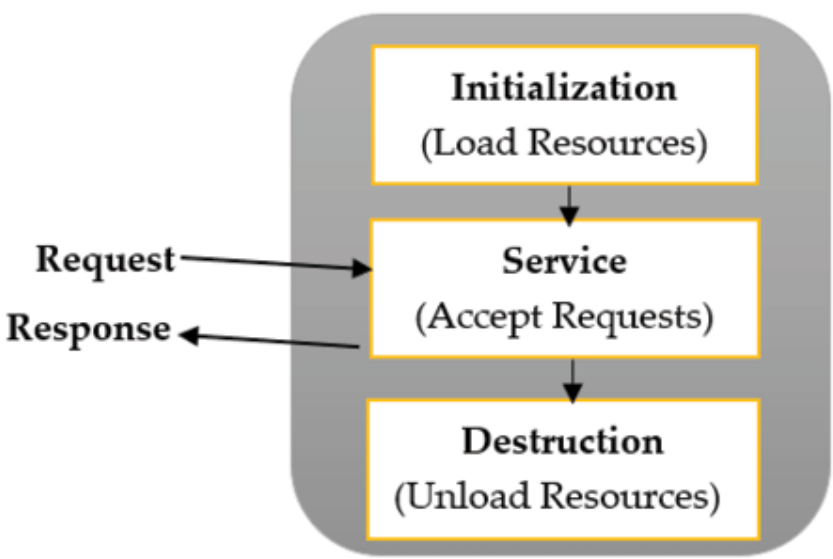


Figure 12.5: Servlet life cycle

Servlet API

The servlet API defines a number of classes and interfaces used for servlet development, provided in the javax.servlet and javax.servlet.http packages. Key interfaces and classes include:

Type	Description
Interface	Servlet interface implemented by all servlets. Defines methods for initialization, handling requests, and destroying the servlet.
Class	GenericServlet class that implements the Servlet interface. Provides a framework for servlets that handle generic (protocol-independent) requests.
Class	HttpServlet extends GenericServlet and provides an abstract class to create servlets that handle HTTP requests (GET, POST, PUT, DELETE, etc.).
Interface	ServletRequest Represents a client request. Provides methods to access request parameters, attributes, and input data.
Interface	HttpServletRequest Extends ServletRequest. Provides HTTP-specific functionality like accessing headers, HTTP methods (GET, POST), and session information.
Interface	ServletResponse Defines the response to be sent to the client. Provides methods to send content, status codes, and set headers.
Interface	HttpServletResponse Extends ServletResponse. Provides HTTP-specific response methods for sending headers, cookies, and HTTP status codes.
Interface	ServletConfig Provides servlet configuration information (such as initialization parameters) to the servlet.
Interface	ServletContext Provides a way to communicate with the servlet container. Allows access to application-wide resources and sharing of data between servlets.
Interface	RequestDispatcher Represents an object that can forward a request from one servlet to another resource (e.g., another servlet, JSP, or HTML file).
Class	ServletException Signals an exception that a servlet can throw to indicate that a problem occurred while processing the request.
Class	Cookie Represents an HTTP cookie. Allows a servlet to manage cookies that store client-specific data.
Interface	HttpSession Provides a way to track user sessions between requests. Enables session management for users.

Table 12.2: Commonly used classes and interfaces of Servlet API

HttpServlet

HttpServlet is the base class that all servlets handling HTTP requests should extend. It is specifically designed to handle HTTP requests and responses in web applications. It is part of the Java Servlet API and extends the generic GenericServlet class, providing additional

functionalities for HTTP-specific processing. The HttpServlet class is located in the **javax.servlet.http** package. It provides methods to handle HTTP requests and responses, making it easier to create web applications that interact with clients via the HTTP protocol. The HttpServlet class includes methods for the most common HTTP methods, like **doGet()**, **doPost()**, **doPut()**, etc., allowing developers to override them for specific processing. Here is a table listing the key methods of the HttpServlet class:

Method	Description
doGet(HttpServletRequest req, HttpServletResponse resp)	Handles HTTP GET requests. Used to retrieve data from the server.
doPost(HttpServletRequest req, HttpServletResponse resp)	Handles HTTP POST requests. Typically used for submitting form data or uploading files to the server.
doPut(HttpServletRequest req, HttpServletResponse resp)	Handles HTTP PUT requests. Used to update or create a resource on the server.
doDelete(HttpServletRequest req, HttpServletResponse resp)	Handles HTTP DELETE requests. Used to delete a resource on the server.
doHead(HttpServletRequest req, HttpServletResponse resp)	Handles HTTP HEAD requests. Similar to GET but only returns headers, not the response body, typically for testing purposes.
doOptions(HttpServletRequest req, HttpServletResponse resp)	Handles HTTP OPTIONS requests. Returns the supported HTTP methods for the resource.
doTrace(HttpServletRequest req, HttpServletResponse resp)	Handles HTTP TRACE requests. Used for debugging to trace the request path.
service(HttpServletRequest req, HttpServletResponse resp)	Dispatches requests to the appropriate doXXX() methods (e.g., doGet, doPost) based on the HTTP method.
init()	Initializes the servlet. Called once when the servlet is instantiated.
destroy()	Cleans up resources before the servlet is destroyed.
getLastModified(HttpServletRequest req)	Returns the last modified time of the requested resource, useful for caching.
doPatch(HttpServletRequest req, HttpServletResponse resp)	Handles HTTP PATCH requests. Used to apply partial modifications to a resource.

***Table 12.3**: List of methods of HttpServlet class*

All the http methods use objects of **HttpServlet** and **HttpSevletResponse**. **HttpServletRequest** provides methods to obtain request parameters, headers, and attributes. It represents the client's request. **HttpServletResponse** allows the servlet to formulate a response, set status codes, headers, and write output content.

The following is a simple example of a servlet named **HelloWorldServlet** that handles GET requests and outputs a basic HTML message:

```
1. import java.io.IOException;
2. import javax.servlet.ServletException;
3. import javax.servlet.http.HttpServlet;
```

```
4. import javax.servlet.http.HttpServletRequest;
5. import javax.servlet.http.HttpServletResponse;
6.
7. public class HelloWorldServlet extends HttpServlet {
8.     @Override
9.     protected void doGet(HttpServletRequest request, HttpServletRespon
10.             throws ServletException, IOException {
11.         response.setContentType("text/html");
12.         response.getWriter().println("<h1>Hello, World!</
    h1>");
13.     }
14. }
```

This servlet handles HTTP GET requests and sends a simple HTML response back to the client.

It is important to understand how to execute a servlet in various scenarios to deploy and test it effectively.

For executing a servlet, a web server would be required. Apache Tomcat is a widely used web container. You can download and install Apache Tomcat, to run servlets. You can also use some IDE like Eclipse or IntelliJ IDEA, which comes with the inbuilt web server.

In an IDE such as Eclipse or IntelliJ IDEA, create a "Dynamic Web Project". Add your **HelloWorldServlet** class in the **.src** folder under a package. Ensure that **web.xml** is correctly configured to map the servlet. For example, the following listing shows a sample **web.xml**:

```
1. <servlet>
2.     <servlet-name>HelloWorldServlet</servlet-name>
3.     <servlet-class>com.example.HelloWorldServlet</servlet-
    class>
4. </servlet>
5. <servlet-mapping>
6.     <servlet-name>HelloWorldServlet</servlet-name>
7.     <url-pattern>/hello</url-pattern>
8. </servlet-mapping>
```

Deploy the project on the Tomcat server. You can start the server from your IDE or the Tomcat command line. Open a web browser and access the servlet at:

http://localhost:8080/YourProjectName/hello

You should see the output:

<h1>Hello, World!</h1>

HttpServletRequest

HttpServletRequest is a key component of the Java Servlet API, specifically designed for handling HTTP requests in web applications. It provides methods to access various aspects of a client's request, such as parameters, headers, attributes, and session information. Understanding HttpServletRequest is essential for building dynamic web applications using servlets. The HttpServletRequest class is part of the **javax.servlet.http** package. It represents the HTTP request sent by the client to the server, encapsulating all relevant data and methods to retrieve that data. Here is a list of methods provided by HttpServletRequest interface:

Method	Description
setAttribute(String name, Object value)	
getAttribute(String name)	associated with the request.
getParameter(String name)	specified parameter as a String (commonly from form submissions or URL query strings).
getParameterValues(String name)	an array of String values for a specified parameter (useful for checkboxes).
getHeader(String name)	specific request header.
getHeaders(String name)	all values associated with a given header.
getHeaderNames()	ation of all the header names in the request.
getMethod()	HTTP method (e.g., GET, POST, PUT, DELETE) used for the request.
getRequestURI()	the request URL that specifies the resource.
getRequestURL()	lete URL used to make the request.
getContextPath()	xt path of the web application.
getSession()	session associated with the request, creating one if it doesn't exist.
getSession(boolean create)	, with an option to create a new one if none exists.
getCookies()	array of Cookie objects sent with the request.
setCharacterEncoding(String enc)	body.
getCharacterEncoding()	oding of the request.

***Table 12.4**: List of methods of HttpServletRequest*

HttpServletResponse

The HttpServletResponse interface is part of the Servlet API and is used by a servlet to craft an HTTP response to the client. It provides methods to manage the content of the response, including headers, cookies, status codes, and the actual body of the response (such as HTML, JSON, etc.). Here is a list of commonly used methods of HttpServletResponse and their descriptions:

Method	Description
void addCookie(Cookie cookie)	Adds a cookie to the response. This method allows you to send cookies to the client.
boolean containsHeader(String name)	Checks if a specified header has been set in the response. Returns true if the header is present, otherwise false.
String encodeRedirectURL(String url)	Encodes a URL for redirection, including session tracking if enabled (i.e., if the client does not support cookies). The session ID will be appended to the URL.
String encodeURL(String url)	Encodes a URL by including the session ID, if necessary, for session tracking. Typically used in the response for URLs embedded in the HTML (e.g., hyperlinks).
void sendError(int sc)	Sends an error response to the client with a specified status code (e.g., 404 for not found, 500 for internal server error).
void sendError(int sc, String msg)	Sends an error response with a specific status code and a custom message.
void sendRedirect(String location)	Redirects the client to a different URL. The URL can be relative to the context or absolute. After this call, the response is considered complete, and no further content is sent.
void setDateHeader(String name, long date)	Sets a header with a specified name and date value. Date is expressed in milliseconds since the epoch (Jan 1, 1970, 00:00:00 GMT).
void addDateHeader(String name, long date)	Adds a header with the specified name and date value (milliseconds since the epoch), appending it to any existing headers with the same name.
void setHeader(String name, String value)	Sets a header with a specified name and value. If the header already exists, this method overwrites the current value.
void addHeader(String name, String value)	Adds a header with a specified name and value. If the header already exists, this method adds another value for the same header.
void setIntHeader(String name, int value)	Sets a header with an integer value. If the header already exists, this method overwrites the current value.
void addIntHeader(String name, int value)	Adds a header with an integer value. If the header already exists, this method appends the new value.
void setStatus(int sc)	Sets the status code for the HTTP response (e.g., 200 for success, 404 for not found). This method should be called before writing any response body.
void setContentType(String type)	Sets the content type of the response (e.g., "text/html" for HTML content or "application/json" for JSON data). The content type is used to inform the client how to interpret it.
void setCharacterEncoding(String charset)	Sets the character encoding (e.g., "UTF-8") for the response body. This is crucial when returning text data that might contain special characters or different languages.
PrintWriter getWriter()	Returns a PrintWriter object that can be used to write character data (like HTML, plain text) to the response body.

`ServletOutputStream getOutputStream()`	Returns a `ServletOutputStream` for writing binary data to the response body (like images, files, or binary data).
`void flushBuffer()`	Forces any content in the response buffer to be sent to the client.
`void reset()`	Clears the response buffer and resets the response status code and headers. Useful when you want to start over after initially sending a response.
`void resetBuffer()`	Clears the content of the response buffer but maintains the status and headers.
`int getBufferSize()`	Returns the size of the response buffer in bytes.
`void setBufferSize(int size)`	Sets the size of the response.
`boolean isCommitted()`	Checks if the response has already been committed (i.e., whether any part of the response has already been sent to the client). If true, changes can't be made to the response.
`void setContentLength(int len)`	Sets the length of the response body in bytes. This helps the client know how much data to expect.
`void setContentLengthLong(long len)`	Sets the length of the response body in bytes, for larger responses.

***Table 12.5**: List of methods of HttpServletResponse*

Let us see one example scenario designed to handle HTTP POST requests, typically when a user submits a form containing profile information such as username and email:

```
1. import java.io.IOException;
2. import javax.servlet.ServletException;
3. import javax.servlet.http.HttpServlet;
4. import javax.servlet.http.HttpServletRequest;
5. import javax.servlet.http.HttpServletResponse;
6.
7. public class UserProfileServlet extends HttpServlet {
8.     @Override
9.     protected void doPost(HttpServletRequest request, HttpServletRespon
10.             throws ServletException, IOException {
11.         // Retrieve parameters from the request
12.         String username = request.getParameter("username");
13.         String email = request.getParameter("email");
14.
15.         // Set an attribute for use in the response
16.         request.setAttribute("user", username);
17.
18.         // Retrieve and display request headers
19.         String userAgent = request.getHeader("User-Agent");
20.
21.         // Processing logic here (e.g., saving user data)
```

```
22.
23.         response.setContentType("text/html");
24.         response.getWriter().println("<h1>Profile for " + username + "<
    h1>");
25.         response.getWriter().println("<p>Email: " + email + "</
    p>");
26.         response.getWriter().println("<p>User-
    Agent: " + userAgent + "</p>");
27.     }
28. }
```

Let us assume there is a simple HTML form that allows the user to submit their profile information, such as username and email:

```
1. <form action="/UserProfileServlet" method="POST">
2.     <label for="username">Username:</label>
3.     <input type="text" id="username" name="username"><br>
4.     <label for="email">Email:</label>
5.     <input type="email" id="email" name="email"><br>
6.     <input type="submit" value="Submit">
7. </form>
```

In both Eclipse and IntelliJ, go to the WebContent (or **src/main/webapp**) folder. Right-click and select New | HTML File. Write above HTML code and name it profile.html. Open a browser and navigate to **http://localhost:8080/[your-project-name]/profile.html**. When the user fills out the form and clicks the **"Submit"** button, the browser sends an HTTP POST request to the server, specifically to the **/UserProfileServlet** URL. The form fields (username and email) are sent as parameters along with the request. Once the form is submitted, the **UserProfileServlet** is invoked. The servlet may already be initialized and loaded by the server (e.g., Apache Tomcat). If it is the first time the servlet is called, the server creates an instance of **UserProfileServlet** and calls its **init()** method (inherited from HttpServlet). However, in most typical scenarios, the servlet will be ready to handle incoming requests. The servlet's **doPost()** method is invoked because the form used the POST method to send data. The servlet retrieves the form data (parameters) sent by the client using the **getParameter()** method:

- **username**: This holds the value entered by the user in the **"Username"** field.
- **email**: This holds the value entered by the user in the

"**Email**" field.

These values are part of the POST request body and can be accessed using **HttpServletRequest**. The servlet sets an attribute in the **HttpServletRequest** object using the **setAttribute()** method. This attribute can be passed on to other resources like JSP pages if needed (though it is not used further in this example). Attributes are typically used for forwarding or sharing data between servlets or JSPs during request dispatching. The servlet retrieves the value of the "**User-Agent**" header from the HTTP request. This header contains information about the client's browser and operating system. The servlet prepares an HTML response using **HttpServletResponse**. It sets the Content-Type to text/html and writes the output to the response. The **response.setContentType()** method informs the client that the content being returned is HTML. The **response.getWriter().println()** methods output the HTML content directly. The browser will render this content as part of the HTTP response. Once the servlet processes the request and sends the response back to the browser, the user will see something like this:

<h1>Profile for john_doe</h1>

** Email: john@example.com **

** User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/93.0.4577.63 Safari/537.36 **

Writing service methods

In a servlet, the service method is responsible for handling client requests and generating responses. The service method of a servlet plays a critical role in processing incoming requests, interacting with external resources, and sending appropriate responses back to the client. For an HTTP-based servlet (**HttpServlet**), the service is generally implemented using specific methods such as **doGet()**, **doPost()**, **doPut()**, **doDelete()**, and other HTTP methods. Each of these methods processes different types of HTTP requests. The general pattern for a service method in a servlet involves three main steps. First step is extracting information from the request. This includes retrieving request parameters, headers, cookies, or session data. Second step is processing the request. This may involve accessing external resources like databases or performing any

business logic. Finally populating the response. Set response headers, retrieve an output stream, and write the response body (such as HTML or JSON). For HttpServlet-based servlets, the service is commonly implemented in the following HTTP methods:

- **doGet()**: Handles HTTP GET requests.
- **doPost()**: Handles HTTP POST requests.
- **doPut()**: Handles HTTP PUT requests.
- **doDelete()**: Handles HTTP DELETE requests.

Below given is an example structure of **doGet()** method:

```
1. @Override
2. protected void doGet(HttpServletRequest request, HttpServletResponse
3.         throws ServletException, IOException {
4.     // Step 1: Extract information from the request
5.     String user = request.getParameter("username");
6.     // Step 2: Process the request (e.g., interact with database or busines
7.     // For example, retrieving user information from a database could ha
8.     // Step 3: Populate the response
9.     response.setContentType("text/html");
10.    PrintWriter out = response.getWriter();
11.    // Send HTML response to client
12.    out.println("<html><body>");
13.    out.println("<h1>Hello, " + user + "</h1>");
14.    out.println("</body></html>");
15. }
```

Before writing anybody content, the servlet must set the response headers (e.g., Content-Type, Content-Length, or custom headers). For example, setting the content type for HTML output:

response.setContentType("text/html");

After setting headers, the response body can be sent using either **PrintWriter** or **ServletOutputStream**. **PrintWriter** is used for sending character-based data such as HTML or plain text. **ServletOutputStream** is used for sending binary data such as images, files, or byte streams. For example:

PrintWriter out = response.getWriter();
out.println("<h1>Response Content Here</h1>");

Once headers are set and the output stream is retrieved, the servlet

writes the body content (the main content of the response) using the **PrintWriter** or **ServletOutputStream**. In cases where an error occurs during processing, the servlet can send an error response using **response.sendError()**.

Example:

response.sendError(HttpServletResponse.SC_NOT_FOUND, "Resource not found");

A more complex scenario involving form data submission using the POST method might look like this:

```
@Override
protected void doPost(HttpServletRequest request, HttpServletResponse
        throws ServletException, IOException {
        // Step 1: Extract form parameters from the request
    String username = request.getParameter("username");
    String email = request.getParameter("email");
        // Step 2: Process the data (e.g., save user information in a databa
    // Business logic or database operations would occur here
        // Step 3: Populate the response
    response.setContentType("text/html");
    response.setCharacterEncoding("UTF-8");
      PrintWriter out = response.getWriter();
        // Send HTML response to the client
out.println("<html><body>");
    out.println("<h1>Hello, " + user + "</h1>");
    out.println("</body></html>");
}
```

Handling multi-valued parameter

In a servlet-based web application, you may encounter a scenario where a form contains fields that can accept multiple values, such as a checkbox group or a multi-select list. The servlet needs to handle this type of form submission and process multiple values for a single parameter. Suppose you have a form that allows users to select their favorite hobbies from a checkbox group. Each checkbox corresponds to a hobby, and users can select multiple hobbies. The HTML form will look like this:

```
<form action="HobbyServlet" method="POST">
```

```
    <label>Select your hobbies:</label><br>
    <input type="checkbox" name="hobby" value="Reading"> Readi
    <input type="checkbox" name="hobby" value="Traveling"> Trave
    <input type="checkbox" name="hobby" value="Cooking"> Cookin
    <input type="checkbox" name="hobby" value="Sports"> Sports<
    <input type="submit" value="Submit">
</form>
```

In the servlet, you can retrieve the multiple values selected for the hobby parameter using the **getParameterValues()** method, which returns an array of strings:

```
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;

@WebServlet("/HobbyServlet")
public class HobbyServlet extends HttpServlet {

    protected void doPost(HttpServletRequest request, HttpServletRespon
            throws ServletException, IOException {

        // Set response content type
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        // Get the selected hobbies from the form
        String[] hobbies = request.getParameterValues("hobby");

        // Check if any hobby is selected
        if (hobbies != null && hobbies.length > 0) {
            out.println("<h2>Your Selected Hobbies:</h2>");
            out.println("<ul>");
            for (String hobby : hobbies) {
                out.println("<li>" + hobby + "</li>");
            }
            out.println("</ul>");
```

```
30.         } else {
31.             out.println("<h2>No hobbies were selected.</h2>");
32.         }
33.
34.         out.close();
35.     }
36.
37.     protected void doGet(HttpServletRequest request, HttpServletRespon
38.             throws ServletException, IOException {
39.         doPost(request, response);
40.     }
41. }
```

The form contains checkboxes for hobbies. Each checkbox has the same name (**hobby**) but different values, such as **Reading**, **Traveling**, etc. When the form is submitted, all selected checkbox values will be sent to the server as an array of strings under the parameter name hobby. The **HobbyServlet** handles the form submission by overriding the **doPost()** method.

The **getParameterValues("hobby")** method retrieves all selected values for the hobby parameter as a string array. The servlet checks if the user selected any hobbies and then displays them in a list. If no hobbies are selected, it displays a message indicating that no hobbies were chosen. If the user selects **Reading** and **Sports** as hobbies, the output will be:

<h2>Your Selected Hobbies:</h2>
<ul>
**
Reading
**
**
Sports
**
**
**

State management

Session management is an essential aspect of server-side web development, especially when dealing with multiple users interacting with a website over time. HTTP, the protocol used for communication between clients and servers, is stateless. This means that the server does not automatically remember any information about previous requests once a response is sent to the client. In web applications, it is often necessary to maintain persistent information

about a user across multiple requests, typically referred to as session tracking. Consider a website where a user logs in and performs various tasks, like browsing items or making a purchase. Once the user logs in, the session begins, and it continues until the user logs out or completes the transaction. However, because HTTP is stateless, the server does not remember the user's previous actions. Every new request looks like a fresh interaction, and the server has no way of knowing that the current request is part of the same session. Since many applications require the server to remember information (like user authentication, shopping carts, etc.), session tracking becomes crucial.

Session tracking is the process of keeping track of previous interactions (requests) between a client and a server. It allows the server to maintain a conversation with the client over multiple requests, simulating persistence by remembering who the user is and what actions they have taken.

Since HTTP does not track sessions on its own, web developers need to manage session tracking manually in their servlets. This is typically done through mechanisms like cookies, URL rewriting, or hidden form fields.

- **Cookies**: The server assigns a session ID and stores it in a cookie. When the client makes subsequent requests, the session ID is sent with each request, allowing the server to retrieve the correct session.
- **URL rewriting**: If cookies are disabled, session IDs can be passed as part of the URL. This method appends the session ID to the URL when making requests.
- **Hidden form fields**: In some cases, session IDs can be sent as hidden fields in forms, though this method is less common today.

However, the most common and reliable method is using the HttpSession interface provided by Java servlets. In this chapter, we will learn the session tracking mechanism using HttpSession and cookies.

Consider an e-commerce website where the user logs in, adds items to their shopping cart, and makes a purchase. Without session management, the server would not remember what items were

added to the cart after each request. With session management, the server can maintain this information throughout the user's interactions:

As the user logs in, the server creates a session for the user and stores their login details (e.g., username) in the session object.

```
HttpSession session = request.getSession();
session.setAttribute("username", "JohnDoe");
```

User Adds Items to Cart. The server stores the items selected by the user in the session, so they can view and modify their cart on subsequent requests.

```
session.setAttribute("cart", shoppingCart);
```

Once the user completes the transaction, the server might invalidate the session to clear any sensitive information.

```
session.invalidate();
```

HttpSession

HttpSession is an interface in the Servlet API that allows servlets to store and retrieve information about a user's session across multiple requests. A session is created by the server for each user who interacts with the web application, and it allows the server to maintain state for that user across multiple requests and responses. Data stored in a session object can be shared across multiple servlets or JSPs. Each user gets a unique session identified by a session ID (stored in a cookie or URL rewriting). Data stored in a session is accessible as long as the session is active (until it is invalidated or times out). Common HttpSession methods are listed below:

Method	Description
public Object getAttribute(String name)	Retrieves the object associated with the specified attribute name from the session.
public void setAttribute(String name, Object value)	Binds an object to the session with the specified attribute name and value.
public Enumeration<String> getAttributeNames()	Returns an enumeration of all attribute names bound to the session.
public void removeAttribute(String name)	Removes the attribute with the specified name from the session.
public String getId()	Returns a unique session ID assigned to this session by the server.
public long getCreationTime()	Returns the time (in milliseconds since the epoch) when this session was created.
public long getLastAccessedTime()	Returns the last time a request associated with this session.
public void invalidate()	Invalidates the session and removes all attributes bound to it.
public boolean isNew()	

	Returns true if the session is new, otherwise returns false.
public void setMaxInactiveInterval(int interval)	Specifies the maximum time interval (in seconds) between requests that the session will remain active.
public int getMaxInactiveInterval()	Returns the maximum time that the session will remain active between requests.
public void invalidate()	Invalidates the session, making it no longer valid.
public boolean isNew()	Checks if the session is just created (true if it is new, otherwise false).

***Table 12.6**: Common HttpSession Methods*

Let us see one example of using HttpSession in a servlet: This example demonstrates how to create and manage a user session. It shows how to store user-specific information, such as a username, in the session and retrieve it on subsequent requests.

Create a servlet e.g., **LoginServlet.java**. The **LoginServlet** accepts the username from the login form and creates a session with the username stored in it:

```
1. import java.io.IOException;
2. import javax.servlet.ServletException;
3. import javax.servlet.http.HttpServlet;
4. import javax.servlet.http.HttpServletRequest;
5. import javax.servlet.http.HttpServletResponse;
6. import javax.servlet.http.HttpSession;
7.
8. public class LoginServlet extends HttpServlet {
9.     @Override
10.     protected void doPost(HttpServletRequest request, HttpServletRespon
11.             throws ServletException, IOException {
12.
13.         // Get the username from the login form
14.         String username = request.getParameter("username");
15.
16.         // Create a new session or retrieve an existing one
17.         HttpSession session = request.getSession();
18.
19.         // Store the username in the session
20.         session.setAttribute("username", username);
21.
22.         // Set the session timeout (30 minutes)
23.         session.setMaxInactiveInterval(30 * 60); // 30 minutes
24.
```

```
        // Redirect to WelcomeServlet
        response.sendRedirect("welcome");
    }
}
```

In this session data, we will retrieve from another servlet e.g., **WelcomeServlet.java**. The **WelcomeServlet** displays a welcome message and includes a form to log out. This servlet dynamically generates the HTML content.

```
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class WelcomeServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletRespon
            throws ServletException, IOException {

        // Retrieve the existing session, do not create a new one if it doesn
        HttpSession session = request.getSession(false);

        // Check if session exists and if the username is stored in the sessi
        if (session != null && session.getAttribute("username") !
= null) {
            String username = (String) session.getAttribute("username");

            // Set the response content type
            response.setContentType("text/html");

            // Generate the welcome message and logout button dynamical
            response.getWriter().println("<html><body>");
            response.getWriter().println("<h1>Welcome, " + username +
h1>");
            response.getWriter().println("<form action='logout' method='
            response.getWriter().println("<input type='submit' value='Log
            response.getWriter().println("</form>");
            response.getWriter().println("</body></html>");
        } else {

```

```
            // Redirect to login page if session is null or username is not fou
            response.sendRedirect("login.html");
        }
    }
}
```

Let us write another servlet the **LogoutServlet**, to invalidate the user session and redirects the user back to the login page:

```
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class LogoutServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletRespon
            throws ServletException, IOException {

        // Get the session and invalidate it
        HttpSession session = request.getSession(false); // false to not cre
        if (session != null) {
            session.invalidate(); // Invalidate the current session
        }

        // Redirect to the login page
        response.sendRedirect("login.html");
    }
}
```

The **login.html** file provides a simple form for the user to enter their username, which is submitted to the **LoginServlet**:

```
<html>
<head>
    <title>Login</title>
</head>
<body>
    <h2>Login</h2>
    <form action="login" method="post">
        <label for="username">Username:</label>
```

```
        <input type="text" id="username" name="username" required>
        <input type="submit" value="Login">
    </form>
</body>
</html>
```

Finally, map the servlets to appropriate URL patterns in the **web.xml** file:

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
        http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd»
        version="3.1">

    <servlet>
        <servlet-name>LoginServlet</servlet-name>
        <servlet-class>LoginServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>LoginServlet</servlet-name>
        <url-pattern>/login</url-pattern>
    </servlet-mapping>

    <servlet>
        <servlet-name>WelcomeServlet</servlet-name>
        <servlet-class>WelcomeServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>WelcomeServlet</servlet-name>
        <url-pattern>/welcome</url-pattern>
    </servlet-mapping>

    <servlet>
        <servlet-name>LogoutServlet</servlet-name>
        <servlet-class>LogoutServlet</servlet-class>
    </servlet>

```

```
    <servlet-mapping>
        <servlet-name>LogoutServlet</servlet-name>
        <url-pattern>/logout</url-pattern>
    </servlet-mapping>
</web-app>
```

Let us understand the flow of the example application. The user accesses the login.html page and submits their username. The form action sends the POST request to the **LoginServlet**, which creates a session and stores the username. Once the login is successful, the user is redirected to the **WelcomeServlet**, which retrieves the session and displays the welcome message. The user can also see a "**Logout**" button. When the user clicks the "**Logout**" button, a GET request is sent to the **LogoutServlet**, which invalidates the session and redirects the user back to the login page.

You can specify a maximum inactive interval for the session using the **setMaxInactiveInterval** method. After the specified time, if the user does not send a new request, the session is automatically invalidated. You can call **invalidate()** to manually end a session. This removes all attributes associated with the session and marks the session as no longer valid.

Cookies

Cookies are an essential part of web development that enable stateful interaction between a client and a server, especially in the context of servlet-based web applications. They allow the server to store small pieces of data (key-value pairs) on the client's browser, which can then be used to maintain state across multiple HTTP requests.

A cookie is a small piece of information that is sent from the server to the client and stored in the client's browser. The browser sends this cookie back to the server with every subsequent request to the same domain, allowing the server to remember things like user preferences, authentication tokens, or session IDs. Cookies are of two types:

- **Session cookies**: These cookies are temporary and are deleted when the user closes the browser. They are used for short-term tracking, such as managing a user session while

they are logged into a website.

- **Persistent cookies**: These cookies are stored on the client's device for a specified period (set by the Max-Age or Expires attribute). They are used for long-term tracking, such as remembering a user's preferences or login information even after the browser is closed.

Key cookie attributes are:

- **Name**: The name of the cookie (used to identify it).
- **Value**: The data or value associated with the cookie.
- **Max-Age**: Defines the lifetime of the cookie in seconds. If set to 0, the cookie will be deleted immediately.
- **Domain**: Specifies the domain to which the cookie belongs. The browser will only send the cookie back to this domain.
- **Path**: Specifies the URL path for which the cookie is valid. The cookie is sent only for requests under this path.
- **Secure**: If set, the cookie is only sent over secure (HTTPS) connections.
- **HttpOnly**: If set, the cookie is not accessible via JavaScript (provides protection against cross-site scripting attacks).

In servlets, the **javax.servlet.http.Cookie** class is used to create, read, and manage cookies.

Creating and sending cookies from the server

A cookie is created on the server side and sent to the client by adding it to the HTTP response. Here is an example of how to create a cookie in a servlet and send it to the client:

```
1. import javax.servlet.*;
2. import javax.servlet.http.*;
3. import java.io.IOException;
4.
5. public class CookieExampleServlet extends HttpServlet {
6.
7.     @Override
8.     protected void doGet(HttpServletRequest request, HttpServletRespon
9.         // Create a new cookie
10.        Cookie userCookie = new Cookie("username", "JohnDoe");
```

```
11.
12.         // Set the cookie's lifespan (in seconds)
13.         userCookie.setMaxAge(60 * 60 * 24); // 1 day
14.
15.         // Add the cookie to the response
16.         response.addCookie(userCookie);
17.
18.         // Set response content type
19.         response.setContentType("text/html");
20.         response.getWriter().println("<h1>Cookie Created and Sent</
    h1>");
21.     }
22. }
```

A cookie named username with the value **JohnDoe** is created. The cookie's lifespan is set to 1 day (60 seconds * 60 minutes * 24 hours). The **response.addCookie()** method is used to send the cookie to the client in the HTTP response header. On subsequent requests, the browser automatically sends the stored cookies to the server. The servlet can retrieve these cookies using the **HttpServletRequest** object. Let us see another servlet for retrieving Cookies on subsequent requests:

```
1. import javax.servlet.*;
2. import javax.servlet.http.*;
3. import java.io.IOException;
4.
5. public class ReadCookieServlet extends HttpServlet {
6.
7.     @Override
8.     protected void doGet(HttpServletRequest request, HttpServletRespon
9.         // Retrieve all cookies from the request
10.         Cookie[] cookies = request.getCookies();
11.
12.         response.setContentType("text/html");
13.         if (cookies != null) {
14.             for (Cookie cookie : cookies) {
15.                 // Display cookie name and value
16.                 response.getWriter().println("<p>Cookie Name: " + cookie.
    p>");
17.                 response.getWriter().println("<p>Cookie Value: " + cookie.
    p>");
```

```
            }
        } else {
            response.getWriter().println("<p>No cookies found!</p>");
        }
    }
}
```

The **request.getCookies()** method is used to retrieve all cookies sent by the client. The servlet then iterates through the cookies, retrieving their name and value, and displays them in the response.

Modifying or deleting cookies

To modify a cookie, simply create a new **Cookie** object with the same name as the one you want to modify, set the new value, and resend it to the client. The old cookie is overwritten. To delete a cookie, you set its Max-Age to 0 and resend it:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.IOException;

public class DeleteCookieServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request, HttpServletRespon
        // Create a cookie with the same name as the one you want to del
        Cookie userCookie = new Cookie("username", "");

        // Set its max age to 0 to delete it
        userCookie.setMaxAge(0);

        // Add the cookie to the response (it will now be deleted in the cli
        response.addCookie(userCookie);

        response.setContentType("text/html");
        response.getWriter().println("<h1>Cookie Deleted</h1>");
    }
}
```

ServletContext

The ServletContext is an interface provided by the Java Servlet API that allows a servlet to interact with its web application's environment. It represents a shared resource that provides servlets within a web application access to configuration and application-level information. Each web application running io a server has a single ServletContext associated with it. The ServletContext is available to all servlets in a web application, making it a convenient place to store information or objects that need to be accessible across the application. It is used to hold application-wide information. It allows servlets to share information across the entire web application. It can store initialization parameters specified in the **web.xml** file, which are available to all servlets in the application. It provides access to resources like files, directories, and other web components that are part of the application.

The **ServletContext** is created when the web application starts and is destroyed when the web application shuts down. Servlets can access the **ServletContext** by calling the **getServletContext()** method. It can be used to share data between servlets, set application-level attributes, and read web application configuration. Common methods of **ServletContext** are listed below:

Method	Description
String getInitParameter(String name)	Returns the value of a context initialization parameter specified in the web.xml file.
Enumeration<String> getInitParameterNames()	Returns an enumeration of all the context initialization parameters for this web application.
Object getAttribute(String name)	Retrieves an object stored in the ServletContext.
void setAttribute(String name, Object value)	Binds an object to the ServletContext under a specific name (global data that can be shared between servlets).
void removeAttribute(String name)	Removes an attribute from the ServletContext.
String getContextPath()	Returns the context path (the portion of the request URL that refers to the web application) associated with the ServletContext.
String getRealPath(String path)	Returns the real path on the server's filesystem corresponding to the given virtual path (useful for accessing files/resources).
RequestDispatcher getRequestDispatcher(String path)	Returns a RequestDispatcher object for forwarding a request to another resource (like another servlet or JSP).
InputStream getResourceAsStream(String path)	Returns an InputStream for reading a resource (like a configuration or data file) as a resource from the web application.
Set<String> getResourcePaths(String path)	Returns a set of resource paths within a specific directory of the web application.
String getServerInfo()	Returns information about the servlet container (web server) such as its

	name and version.
ServletContext getContext(String uripath)	Returns the ServletContext object for a specific URL path within the same web application in the same server (if allowed), enabling communication between different web applications.

***Table 12.7**: Common methods of ServletContext*

Example use cases of ServletContext

The ServletContext interface in Java is a powerful tool for sharing data, resources, and configuration settings across the entire web application. Here are some common use cases for ServletContext.

Sharing data across servlets

The ServletContext allows servlets to share global data, such as a counter for tracking active users. We will build a simple web application that tracks the total number of visitors to the site. Every time a user accesses the site (via any servlet), the total number of visitors will be updated and displayed.

Create the **VisitorServlet** servlet. This servlet will use the **ServletContext** to read and update the visitor count. Every time a user visits the **/visitor** URL, the visitor count will increase.

```
1. package com.example;
2.
3. import java.io.IOException;
4. import javax.servlet.ServletException;
5. import javax.servlet.annotation.WebServlet;
6. import javax.servlet.http.HttpServlet;
7. import javax.servlet.http.HttpServletRequest;
8. import javax.servlet.http.HttpServletResponse;
9. import javax.servlet.ServletContext;
10.
11. @WebServlet("/visitor")
12. public class VisitorServlet extends HttpServlet {
13.
14.     @Override
15.     protected void doGet(HttpServletRequest request, HttpServletRespon
16.             throws ServletException, IOException {
17.
18.         // Get the ServletContext object
19.         ServletContext context = getServletContext();
```

```
20.
21.         // Retrieve the visitor count from the context (initialize to 0 if not
22.         Integer visitorCount = (Integer) context.getAttribute("visitorCoun
23.         if (visitorCount == null) {
24.             visitorCount = 0;
25.         }
26.
27.         // Increment the visitor count
28.         visitorCount++;
29.
30.         // Store the updated visitor count back into the context
31.         context.setAttribute("visitorCount", visitorCount);
32.
33.         // Display the visitor count on the webpage
34.         response.setContentType("text/html");
35.         response.getWriter().println("<h1>Welcome to the website!
    </h1>");
36.         response.getWriter().println("<p>Total visitors: " + visitorCount
    p>");
37.     }
38. }
```

In the **VisitorServlet**, the **ServletContext** is retrieved using **getServletContext()**. The visitor count is stored in the context as an attribute, allowing it to be shared between all servlets in the application. The count is initialized to 0 if it is not already set. Each time the servlet is accessed, the visitor count is incremented and updated in the **ServletContext**.

Sharing global configuration parameters

Web application-wide configuration parameters can be defined in web.xml and accessed through the **ServletContext**. You can define global configuration parameters for the entire application using the <**context-param**> tag in the **web.xml** file. The following is the sample **web.xml** file:

```
1. <web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
2.   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
4.   http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd»
5.   version="3.1">
```

```
6.
7.    <!-- Define context parameters -->
8.    <context-param>
9.        <param-name>appName</param-name>
10.       <param-value>My Sample Web Application</param-
value>
11.   </context-param>
12.
13.   <context-param>
14.       <param-name>adminEmail</param-name>
15.       <param-value>admin@example.com</param-value>
16.   </context-param>
17.
18.   <servlet>
19.       <servlet-name>ConfigServlet</servlet-name>
20.       <servlet-class>com.example.ConfigServlet</servlet-
class>
21.   </servlet>
22.
23.   <servlet-mapping>
24.       <servlet-name>ConfigServlet</servlet-name>
25.       <url-pattern>/config</url-pattern>
26.   </servlet-mapping>
27.
28. </web-app>
```

Create the **ConfigServlet** servlet. The servlet will retrieve the configuration parameters defined in the **web.xml** using the **ServletContext** object.

```
1. package com.example;
2.
3. import java.io.IOException;
4. import javax.servlet.ServletException;
5. import javax.servlet.annotation.WebServlet;
6. import javax.servlet.http.HttpServlet;
7. import javax.servlet.http.HttpServletRequest;
8. import javax.servlet.http.HttpServletResponse;
9. import javax.servlet.ServletContext;
10.
11. @WebServlet("/config")
12. public class ConfigServlet extends HttpServlet {
```

```
13.
14.    @Override
15.    protected void doGet(HttpServletRequest request, HttpServletRespon
16.            throws ServletException, IOException {
17.
18.        // Get the ServletContext object
19.        ServletContext context = getServletContext();
20.
21.        // Retrieve the configuration parameters
22.        String appName = context.getInitParameter("appName");
23.        String adminEmail = context.getInitParameter("adminEmail");
24.
25.        // Display the configuration parameters on the webpage
26.        response.setContentType("text/html");
27.        response.getWriter().println("<h1>Application Name: " + appNa
   h1>");
28.        response.getWriter().println("<p>Admin Email: " + adminEmail
   p>");
29.    }
30. }
```

The <**context-param**> tags in **web.xml** are used to define two parameters: **appName** and **adminEmail**. These parameters are accessible throughout the entire web application. In the **ConfigServlet**, the **ServletContext** object is obtained using **getServletContext()**. The **getInitParameter()** method is used to retrieve the values of the parameters (**appName** and **adminEmail**) from the **ServletContext**. When the **/config** URL is accessed, the servlet responds with an HTML page that displays the values of the configuration parameters.

Application Name: My Sample Web Application
Admin Email: admin@example.com

ServletConfig

The **ServletConfig** interface in Java is used to pass configuration information to a servlet at initialization time. Each servlet has a unique **ServletConfig** object that provides access to the servlet's configuration parameters, as defined in the **web.xml** (deployment descriptor) or using annotations. While the **ServletContext** provides configuration that is shared by all servlets in the application, the

ServletConfig is specific to a single servlet. The servlet container creates **ServletConfig** object for each servlet when the servlet is initialized. The servlet container passes the **ServletConfig** object to the servlet's **init()** method when the servlet is being initialized.

The **ServletConfig** interface defines the following methods:

Method	Description
String getInitParameter(String name)	Returns the value of the initialization parameter based on the provided parameter name.
Enumeration<String> getInitParameterNames()	Returns an enumeration of initialization parameter names defined for this servlet.
ServletContext getServletContext()	Returns the ServletContext object, which provides access to shared resources across the application.
String getServletName()	Returns the name of the servlet as defined in web.xml or through annotations.

***Table 12.8**: Methods of ServletConfig*

Here is an example that demonstrates how to use **ServletConfig** in a servlet. Define initialization parameters in **web.xml**.

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
    http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
    version="3.1">

    <servlet>
        <servlet-name>ConfigServlet</servlet-name>
        <servlet-class>com.example.ConfigServlet</servlet-class>
        <!-- Define initialization parameters for this servlet -->
        <init-param>
            <param-name>email</param-name>
            <param-value>support@example.com</param-value>
        </init-param>
        <init-param>
            <param-name>environment</param-name>
            <param-value>production</param-value>
        </init-param>
    </servlet>

```

```
    <servlet-mapping>
        <servlet-name>ConfigServlet</servlet-name>
        <url-pattern>/config</url-pattern>
    </servlet-mapping>

</web-app>
```

Create the servlet (**ConfigServlet**). The servlet will retrieve its own initialization parameters via **ServletConfig** and display them:

```
package com.example;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;

@WebServlet("/config")
public class ConfigServlet extends HttpServlet {

    private String email;
    private String environment;

    @Override
    public void init(ServletConfig config) throws ServletException {
        // Initialize the servlet by reading configuration parameters
        this.email = config.getInitParameter("email");
        this.environment = config.getInitParameter("environment");

        // Log the servlet name for debugging
        String servletName = config.getServletName();
        System.out.println("Initializing " + servletName + " servlet");

        // Optionally, get the ServletContext
        ServletContext context = config.getServletContext();
        // Do something with the ServletContext if needed
    }
```

```
32.
33.     @Override
34.     protected void doGet(HttpServletRequest request, HttpServletRespon
35.             throws ServletException, IOException {
36.         // Display the servlet's configuration parameters
37.         response.setContentType("text/html");
38.         response.getWriter().println("<h1>Servlet Configuration Parame
    h1>");
39.         response.getWriter().println("<p>Email: " + email + "</
    p>");
40.         response.getWriter().println("<p>Environment: " + environment
    p>");
41.     }
42. }
```

The initialization parameters email and environment are specific to the **ConfigServlet**. These are defined inside the <**init-param**> tags within the <**servlet**> element in **web.xml**. In the **init()** method of the servlet, the **ServletConfig** object is passed as an argument. The servlet uses the **getInitParameter()** method to retrieve the initialization parameters. The parameters are stored in instance variables (email and environment) to be used later in the **doGet()** method.

The **getServletName()** method retrieves the name of the servlet (useful for logging or debugging). The **getServletContext()** method gives access to the **ServletContext**, which allows interaction with global configurations and shared resources. When a user accesses /**config**, the **doGet()** method is called, which displays the initialization parameters retrieved from the **ServletConfig**.

Conclusion

In conclusion, this chapter has provided an in-depth exploration of servlet-based web development within the J2EE framework. By understanding the servlet lifecycle, the roles of HttpServlet, HttpServletRequest, HttpServletResponse, ServletContext and ServletConfig, and mastering state management techniques, developers are better equipped to create robust, scalable, and dynamic web applications. The chapter also highlighted the importance of separating concerns between different layers in the J2EE architecture, ensuring that web applications are both

maintainable and scalable. Mastery of servlets is a fundamental skill for any developer working within the J2EE environment, laying the groundwork for more advanced topics in enterprise web application development.

In the next chapter, we will learn about Dynamic Web Content with JSP.

Exercises

1. Write a simple servlet that responds with "Hello, World!" when accessed via a web browser.
2. Modify the servlet to accept a name as a query parameter and respond with "Hello, [name]!"
3. Create a servlet that handles a form submission. The form should take user input (e.g., name and email) and display a confirmation message back to the user.
4. Validate the email format before displaying the confirmation.
5. Implement a servlet that tracks user visits using HTTP sessions. Each time the user accesses the servlet, increment a visit counter and display the total number of visits.
6. Add functionality to reset the counter via a button.
7. Write a servlet that reads configuration parameters from the web.xml file and displays them in the browser.
8. Allow users to modify these parameters through a form and save changes.
9. Write a servlet that stores user information (like name and email) in the ServletContext. Create another servlet that retrieves and displays this information.
10. Create a servlet that sets a list of available products as an attribute in the ServletContext. Create another servlet that retrieves this list and displays it in an HTML table.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the

Authors:

https://discord.bpbonline.com

Chapter 13
Dynamic Web Content with JSP

Introduction

In this chapter, we explore dynamic web content with **JavaServer Pages (JSP)**, which is a core component of the **Java 2 Enterprise Edition (J2EE)** framework. JSP provides an efficient way to develop web applications by enabling the creation of dynamic web pages using both static HTML and dynamic Java code. By integrating Java code within HTML, JSP simplifies the development of web-based applications, allowing developers to generate dynamic content such as user-specific responses or database-driven data. This chapter focuses on understanding the fundamental concepts of JSP, its lifecycle, directives, actions, and how it integrates with Servlets to create scalable and maintainable web applications.

Structure

The chapter covers the following topics:

- Introduction to JSP
- Comparison with servlet

- JavaServer Pages architecture
- JavaServer Pages life cycle
- JSP standard syntax
- JSP implicit objects
- Integration of JSP with servlet
- JSP Standard Tag Libraries
- JSP session management

Objectives

The objective of this chapter is to provide a thorough understanding of JSP for creating dynamic web content. It focuses on explaining the JSP architecture, its lifecycle, and how it simplifies the development of web applications by integrating Java code into HTML. The chapter also aims to highlight the differences between JSP and servlets, the usage of JSP directives and actions, and the integration of JSP with servlets to follow the **Model-View-Controller** (**MVC**) pattern. Additionally, it covers the use of **JSP Standard Tag Libraries** (**JSTL**) for writing cleaner, more maintainable code and discusses various methods of session management to maintain state across multiple client requests.

Introduction to JSP

JSP is a technology that helps developers create dynamically generated web pages based on HTML, XML, or other document types. JSP is built on top of the Java Servlet technology and allows embedding Java code directly into web pages. Its goal is to make the development of dynamic web content simpler by providing a clear separation between presentation and business logic. At its core, JSP allows developers to write HTML (or other content) alongside Java code in the same file. When a JSP page is requested by the client, it is first compiled into a servlet by the web container. This servlet processes the request, executes the embedded Java code, and generates a dynamic response (usually in the form of an HTML page).

JSP offers several features that make it powerful and suitable for

dynamic web content generation:

- **Separation of concerns**: JSP allows a clear separation of presentation logic (HTML) from business logic (Java code). While Java code can be written directly on a JSP page, it is typically better to separate it into JavaBeans or other backend components.
- **Tag libraries**: JSP supports custom tag libraries (e.g., JSTL) that allow developers to implement custom functionality without writing Java code directly into JSP pages.
- **Integration with Java EE**: JSP works seamlessly with other Java EE technologies, including servlets, filters, and JavaBeans, providing a comprehensive web development platform.
- **Simplified coding**: JSP reduces the amount of boilerplate code compared to servlets, as it eliminates the need to explicitly write **HttpServletRequest** and **HttpServletResponse** handling logic.

Comparison with servlet

Both JSP and Servlets are key components of Java's web development framework. They have a similar purpose (handling web requests and generating responses), but they differ in their approach and specific use cases.

Code-structure and development style

A servlet is a Java class that extends the **HttpServlet** class. The entire HTML content must be generated inside Java methods like **doGet()** or **doPost()** using output streams (**PrintWriter**).

Example:

```
1. response.setContentType("text/html");
2. PrintWriter out = response.getWriter();
3. out.println("<html><body>");
4. out.println("<h1>Hello, World!</h1>");
5. out.println("</body></html>");
```

Managing HTML in servlets can become cumbersome as the complexity of the page increases.

JSP is an HTML-based file with Java code embedded directly. The HTML content can be written normally, and Java code is inserted only where needed. JSP is more suitable for pages with lots of static content mixed with a bit of dynamic data.

Ease of use

Servlets requires more code to handle requests and responses. HTML must be written as string literals, making it harder to manage larger web pages.

JSP is easier for web designers who are more familiar with HTML/CSS. Java code is only added where necessary, making it simpler to handle dynamic data on a primarily static page.

Separation of concerns

In servlets, the presentation (HTML) and business logic (Java) are mixed together, leading to harder-to-maintain code if the application scales. For instance, generating a large HTML form in servlets would require you to handle the entire form in Java code.

JSP encourages a clear separation of concerns by keeping the presentation layer (HTML) separate from the business logic. JavaBeans or custom tags are often used to interact with Java code in the backend, promoting cleaner and more maintainable code.

Performance

Servlets are slightly faster than JSP since servlets are directly written in Java and do not need to go through the additional step of being translated from JSP to a servlet.

JSP is initially slower than servlets because the JSP page is first compiled into a servlet. However, after the initial compilation, both perform similarly since JSPs are essentially servlets at runtime.

Usage in modern web applications

Servlets are best suited for handling backend logic, such as processing form submissions, interacting with databases, and managing sessions. Often used to control application flow (acting as controllers in MVC architectures).

While JSP is best suited for the view layer in web applications,

especially in the MVC architecture. JSP pages are typically used to display dynamic content to the user after the backend logic (handled by servlets) has been executed.

Sample servlet and JSP program:

```
import java.io.* ;
import javax.servlet.* ;
import javax.servlet.http.* ;
public class HelloWorldServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                                HttpServletResponse response)
                        throws IOException, ServletException {
        response.setContentType("text/html") ;
        PrintWriter out = response.getWriter() ;
        out.println("<html><body>") ;
   out.println("<h1>Hello " +
                        request.getParameter("username") + "!</
h1>") ;
        out.println("</html></body>") ;
    }
}
```

An equivalent JSP page:

```
<html> <body>
<h1>Hello <%= request.getParameter("username") %> </
h1>
</body> </html>
```

JavaServer Pages architecture

JSP is built on top of the Java Servlet architecture, and its primary purpose is to simplify the creation of dynamic web content. The JSP architecture involves several components and steps, from the time a client sends a request to the web server until the JSP generates the response. This process can be broken down into several stages, including translation, compilation, execution, and response generation. At the heart of JSP architecture is the concept that JSP is an abstraction built on top of Servlet technology. When a JSP page is requested, it is first compiled into a servlet, which handles the request-response cycle. Here is an overview of how a JSP is handled by the server.

A client (browser) sends an HTTP request for a JSP page. The server receives the request and forwards it to the JSP container. If the JSP has never been requested before, it is translated into a servlet. The servlet is then compiled into Java bytecode. The JSP-generated servlet processes the request and generates a response. The server sends the generated response (typically HTML) back to the client.

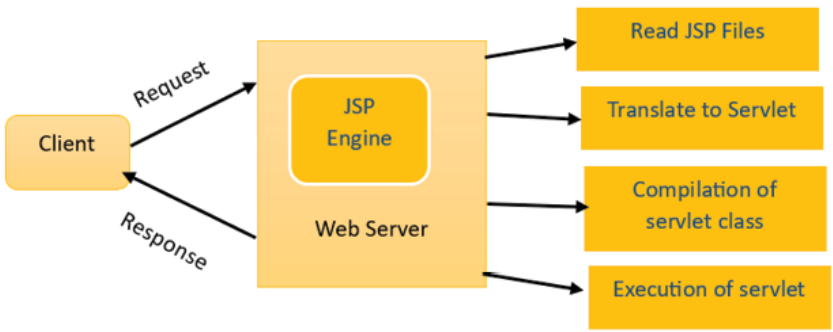


***Figure 13.1**: JSP handling by a web server*

The JSP architecture relies on several components to manage the lifecycle and processing of JSP requests:

- **Client (Browser)**: The client sends an HTTP request to the web server, typically by entering a URL in the browser or interacting with a form. This request might include form data (parameters), query strings, cookies, or session information.
- **Web server**: The web server receives the client's HTTP request and is responsible for forwarding it to the appropriate JSP container or servlet container (e.g., Apache Tomcat, Jetty). The web server acts as a bridge between the client and the JSP/Servlet container.
- **JSP container**: The JSP container is part of the servlet container (such as Apache Tomcat or Jetty). It handles JSP pages specifically. When the web server receives a request for a JSP file, the JSP container manages the lifecycle of the JSP page, including translating the JSP file into a servlet, compiling the servlet, executing the servlet (if already compiled), Managing the HTTP request and generating a dynamic response. The JSP container ensures that the JSP-

to-servlet translation happens transparently without manual intervention.

- **JSP compiler**: When a client requests a JSP for the first time (or when the JSP has been updated), the JSP container hands over the JSP page to the JSP compiler. The compiler translates the JSP into Java source code, specifically into a servlet that can process HTTP requests. This is an essential part of the JSP architecture, which effectively allows JSPs to behave like servlets.

Servlet container

Once the JSP has been compiled into a servlet, the servlet container takes over. It loads the generated servlet class, and from this point on, the servlet lifecycle is managed. The servlet container is responsible for initializing the servlet, managing multiple client requests via multithreading, and destroying the servlet when no longer needed.

JavaServer Pages life cycle

The JSP life cycle is a series of phases that a JSP page goes through, from its creation to its destruction. The JSP life cycle consists of several phases. The first phase is the **translation phase**. In this phase, the JSP page is translated into a Java servlet. The JSP compiler checks the syntax of the JSP page and converts it into a Java servlet. The second phase is compilation phase. In this phase, the Java servlet (generated in the translation phase) is compiled into a .class file. The next phase is the loading phase. In this phase, the servlet container (e.g., Apache Tomcat) loads the compiled servlet class into memory. Then comes the **instantiation phase**, where an instance of the servlet class is created by servlet container for each request. The next phase is the initialization phase; the servlet instance is initialized. The **init()** method is called, which is used to initialize the servlet. Now, the servlet instance is ready for request handling. In the request handling phase, the servlet instance handles the HTTP request. The servlet's **service()** method processes the client's request, executes any business logic, and generates the response (usually in the form of HTML). The **service()** method is

called, which is responsible for processing the request and generating the response. The final phase is **destruction phase**. In this phase, the servlet instance is destroyed. The **destroy()** method is called, which is used to release any resources held by the servlet.

The translation and compilation phases only take place the first time a JSP page is requested or when the JSP page is modified. The initialization phase, where the **init()** method is called, takes place once when the servlet is loaded for the first time into memory. This occurs after the servlet class has been instantiated. Subsequent requests bypass this phase, as the servlet is already initialized and ready to handle incoming requests.

JSP life cycle is managed through different life cycle methods. The following methods are called during the JSP life cycle:

jspInit()

In JSP, the **jspInit()** method is an important lifecycle method that is executed first when a JSP page is loaded. This method is ideal for performing any JSP-specific initialization tasks, such as setting up database connections, loading resources, or initializing application-level variables. The **jspInit()** method is defined within the JSP declaration element, which is denoted by <%! ... %>. This declaration element allows developers to declare methods and variables that are available throughout the JSP's lifecycle. For example, a typical declaration might look like this:

```
<%!
public void jspInit() {
    // Initialization code, such as setting up database connections
}
%>
```

It is important to note that **jspInit()** is called only once during the lifetime of the JSP component, making it an efficient place to initialize resources that will be used across multiple requests. This ensures that costly initialization processes are not repeated for every user request, enhancing performance and resource management.

_jspService()

The **_jspService(HttpServletRequest request, HttpServletResponse response)** method in JSP is automatically generated by the JSP engine and functions similarly to the service() method in a servlet. It handles each request sent to the JSP page, managing both HTTP GET and POST methods. Unlike servlets, developers never need to manually define or customize this method. Instead, all Java code written within JSP scripting elements (such as <% %> or <%= %>) is automatically inserted into the **_jspService()** method by the JSP engine during the translation phase when the JSP is converted into a servlet. This method is called for every request, and its main role is to process the client's request and generate the appropriate response. While it plays a critical role in handling request-response cycles, it remains hidden from the developer, with the JSP container managing its creation and execution.

jspDestroy()

In JSP, the **jspDestroy()** method serves as the cleanup method that is executed when the JSP page is being taken out of service. This method is particularly useful for releasing resources and performing any necessary cleanup tasks before the JSP component is destroyed. Developers can define the **jspDestroy()** method within a JSP declaration element, indicated by <%! ... %>. For example:

```
<%!
public void jspDestroy() {
    // Cleanup code, such as closing database connections
}
%>
```

By implementing **jspDestroy()**, developers can ensure that any open resources—such as database connections, file handles, or network sockets—are properly released. This is vital for preventing memory leaks and ensuring that system resources are managed effectively, which contributes to the overall performance and stability of the application.

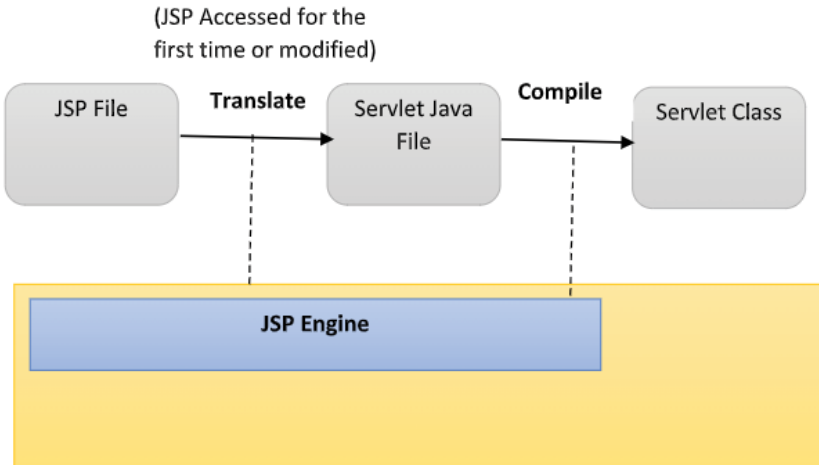


***Figure 13.2**: Processing of JSP page*

JSP standard syntax

A JSP page is structured to seamlessly blend static and dynamic content. The static parts, typically plain HTML, are copied verbatim into the response output in the order they appear. Alongside this, dynamic behavior is added through various components like JSP scriptlets, which embed Java code within <% %> tags, and JSP expressions, which insert the result of a Java expression directly into the response using <%=%>. Directives, enclosed in <%@ %> tags, provide instructions to the JSP container and influence the entire page, while JSP actions in XML syntax, such as <**jsp:include**> and <**jsp:useBean**>, control the page's flow and resource inclusion. Custom tag libraries (**taglibs**) can also be utilized to encapsulate reusable logic, making the page more modular. Besides HTML tag elements, JSP provides four basic categories of constructors (markup tags), comments, scripting elements, directives, and standard actions.

JSP comments

JSP comments are a useful feature that allows developers to include notes and explanations in their JSP files without affecting the output sent to the client. They serve as a means of documentation within the code, making it easier for developers to understand the

logic and structure of the JSP page. The syntax of JSP comment is given below:

<%-- This is a JSP comment --%>

JSP comments are not included in the output that is sent to the client. This means that anything within the <%-- ... --%> tags is purely for the developer's reference and will not be visible in the HTML rendered in the browser. They can span multiple lines, making them suitable for longer explanations or notes.

JSP scripting elements

The scripting elements in JSP provide the ability to insert Java code directly within the JSP file, allowing for dynamic content generation. There are three main types of scripting elements:

- **Scriptlet tag (<% ... %>)**: Scriptlets allow you to execute Java code on the server side. Any code written within the scriptlet tags is processed when the JSP is requested, and you can perform operations like variable declarations, loops, conditionals, etc.

```
<%
   String message = "Hello, World!";
   out.println(message);
%>
```

  In this example, a string variable message is declared, and its value is printed to the output. Another example with more java code is shown below:

```
<html><head></head><body>
Your pets:<p>
<table border cellspacing=0 cellpadding=5>
<%
String [] pets = request.getParameterValues("pets") ;
for (int i = 0 ; i < pets.length ; i++) {
%>
      <tr><td>  <%= pets [i] %>  </td></tr>
<%
}
%>
</table>
</body></html>
```

- **Expression tag (<%= ... %>)**: The expression tag is used to output data directly to the response. The expression inside the tags is evaluated, converted to a string, and inserted into the HTML response stream. For example, the JSP page:

```
<html><head></head><body>
<
% out.println("Now is" + new java.util.Date()); %>
</body></html>
```

This can be rewritten as:

```
<html><head></head><body>
Now is <%= new java.util.Date() %>
</body></html>
```

This outputs the current date and time directly into the HTML response when the JSP is rendered.

- **Declaration tag (<%! ... %>)**: Declarations are used to define variables and methods that can be used throughout the JSP page. Code within the declaration tags is executed when the JSP is compiled, not on each request. This is ideal for defining reusable methods or variables. The following example demonstrates the declaration of a variable and method:

```
<%! int count = 0; %>
<html><head></head><body>
This JSP page has been accessed <%= count++ %> times
</body></html>
```

Note that actually, it does not matter where the declaration element appears on the JSP page, since it is not associated with request handling thread of control. A JSP declaration element can include a method declaration, e.g.:

```
<%!
    String myMethod() {
        return "madness";
    }
%>
<html><head></head><body>
There is <%= myMethod() %> in my method.
</body></html>
```

Java Server Pages directives

In JSP, directives are special instructions given to the JSP engine that influence the entire page's processing. These directives are not directly visible in the generated HTML but affect the structure and behavior of the JSP page during translation time when the JSP is converted into a servlet. There are three main types of JSP directives, page directive, include directive, taglib directive. Let us look at each one in detail:

- **Page directive**: The page directive defines page-dependent attributes that affect the entire JSP page. It usually defines properties such as error handling, session tracking, content type, and more. This directive is typically declared at the top of the JSP file. Syntax of page directive is as follows:

  <%@ page attribute="value" %>

  Page directive has many attributes. Common attributes are listed below:

	Description	
<%@page language="java" %>	Specifies the scripting language for the page. Default is java.	
<%@page extends="com.example.MyServlet" %>	Specifies the superclass of the servlet generated from the JSP page.	
<%@page import="java.util.*,java.io.*" %>	Imports Java classes or packages to be used within the JSP page. Multiple classes can be imported with commas.	
<%@page session="false" %>	Specifies whether the page participates in HTTP session tracking. Default is true.	
<%@page buffer="16kb" %>	Defines the size of the output buffer. It can be a number followed by kb or none.	
<%@page autoFlush="true" %>	Indicates whether the buffer should automatically flush when full (true) or throw an exception (false).	
<%@page isThreadSafe="false" %>	Indicates if the page is thread safe. Default is true.	
<%@page errorPage="errorPage.jsp" %>	Specifies a URL to redirect to if an uncaught exception occurs.	
<%@page isErrorPage="true" %>	Indicates if the page is an error page that can handle exceptions.	
<%@page contentType="text/html; charset=UTF-8" %>	Sets the MIME type and character encoding for the response. Defaults to text/html and ISO-8859-1.	
<%@page pageEncoding="UTF-8" %>	Specifies the character encoding for the JSP page itself.	
<%@page info="My JSP Page" %>	Provides information about the JSP page, which can be retrieved by getServletInfo() method of the generated servlet.	
<%@page isELIgnored="true" %>	Specifies whether to ignore Expression Language (EL) within the JSP. Default is false.	

***Table 13.1**: Key attributes of the JSP page directive*

This table provides an overview of the key attributes of the JSP

page directive with corresponding examples to demonstrate their usage.

Include directive

The include directive allows you to include another JSP file or static content into the current JSP page at translation time. This is different from the **<jsp:include>** action, which includes a resource at request time. The general syntax is as follows:

<%@ include file="relativeURL" %>

For example, to include header.jsp file, the statement will be as follows:

<%@ include file="header.jsp" %>

File specifies the relative path of the file to include. It can be a JSP file, HTML file, or even a text file. The content of the included file becomes part of the current JSP page at translation time, which means any changes to the included file will require the main JSP to be recompiled. The included file can contain static content or JSP code. It is useful for reusing common parts of the webpage, like headers, footers, or navigation bars.

Taglib directive

The **taglib** directive is used to declare a tag library that contains custom tags used in the JSP page. Tag libraries allow you to define reusable code and extend the functionality of JSP by creating custom tags. The syntax of **taglib** directive is as follows:

<%@ taglib uri="uri" prefix="prefix" %>

For example,

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

uri refers to the **Uniform Resource Identifier (URI)** that uniquely identifies the **tag library descriptor (TLD)** file. This URI is either a standard URI for a common library like JSTL or a custom URI for user-defined tags. **prefix**: defines a prefix for the custom tags defined in the tag library. The prefix is used to refer to the custom tags on the JSP page.

Example of using a custom tag:

```
<c:if test="${user != null}">
   Welcome, ${user.name}!
</c:if>
```

In this example, **c** is the prefix associated with JSTL core tags, and if is a custom tag used for conditional logic.

JavaServer Pages actions

JSP actions are special XML tags in JSP that allow you to control the behavior of the servlet container or perform some specific tasks, like including content, forwarding requests, or interacting with JavaBeans. They are designed to simplify common tasks in JSP development. The following are some common JSP actions:

- **<jsp:include>**: The **<jsp:include>** action is used to include the content of another resource (such as another JSP, HTML, or servlet) during request processing. Unlike the include directive (**<%@ include %>**), which includes a file at translation time, **<jsp:include>** is processed at request time, meaning it dynamically includes content. The general syntax of **<jsp:include>** is given below:

  <jsp:include page="relativeURL" flush="true" />

  This action has two attributes:

  page: The relative URL of the resource to include.

  flush: Specifies whether the output buffer should be flushed before including the resource. It is usually set to true.

  Observe the following example:

  <jsp:include page="header.jsp" />

  In this example, the content of header.jsp will be dynamically included when the JSP is requested.

- **<jsp:forward>**: The **<jsp:forward>** action forwards the current request to another resource (JSP, servlet, or HTML). After the forward, the control is transferred, and no further processing occurs on the original page. The general syntax is as follows:

  <jsp:forward page="relativeURL" />

  It has page attribute which contains the relative URL of the resource to which the request is forwarded. Observe the

below given example:

```
<jsp:forward page="login.jsp" />
```

In this example, the current request is forwarded to login.jsp.

- **<jsp:param>**: The **<jsp:param>** action is used to pass parameters to an included or forwarded resource, or within a **<jsp:plugin>** action. It allows you to pass key-value pairs as part of the request. The general syntax is:

  ```
  <jsp:param name="paramName" value="paramValue" />
  ```

  For example:

  ```
  <jsp:forward page="searchResults.jsp">
      <jsp:param name="query" value="Java" />
  </jsp:forward>
  ```

  In this example, the query parameter with value **Java** is passed to the **searchResults.jsp** page during forwarding.

- **<jsp:plugin>**: The **<jsp:plugin>** action is used to embed a Java applet or JavaBean in the generated HTML page. This action helps in generating browser-specific tags (like **<object>** for Internet Explorer and **<embed>** for other browsers) to run Java applets or JavaBeans. The general syntax is as follows:

  ```
  <jsp:plugin type="bean|applet" code="className"
  codebase="URL" width="widthValue"
  height="heightValue">
      </jsp:plugin>
  ```

  It has various attributes. type specifies whether the plugin is a bean or an applet. code attribute has the fully qualified class name of the applet or JavaBean. codebase specifies the URL where the Java applet or bean is located. width and height define the dimensions of the plugin. The following example demonstrates its use:

  ```
  <jsp:plugin type="applet" code="com.example.MyApplet"
  width="300" height="300">
  </jsp:plugin>
  ```

  This will generate the necessary HTML tags to embed the Java applet in the page.

JavaBeans with JSP

The other three action tags are related to JavaBean component. A JavaBean is a reusable software component written in Java that follows specific conventions. It is a simple Java class that follows specific conventions and is primarily used for encapsulating data. It provides getter and setter methods to access private properties (fields) and can be used in JSP pages to share data between requests, sessions, or applications. JavaBeans are often used in web applications, particularly in JSP, to handle and store application data.

Example of a simple JavaBean:

```
1. package com.example;
2.
3. import java.io.Serializable;
4.
5. public class UserBean implements Serializable {
6.     private String name;
7.     private String email;
8.
9.     // No-argument constructor
10.    public UserBean() {}
11.
12.    // Getter for name
13.    public String getName() {
14.        return name;
15.    }
16.
17.    // Setter for name
18.    public void setName(String name) {
19.        this.name = name;
20.    }
21.
22.    // Getter for email
23.    public String getEmail() {
24.        return email;
25.    }
26.
27.    // Setter for email
```

```
    public void setEmail(String email) {
        this.email = email;
    }
}
```

You can interact with JavaBeans in JSP using action tags like **<jsp:useBean>**, **<jsp:setProperty>**, and **<jsp:getProperty>**.

<jsp:useBean>

<jsp:useBean> action tag is used to instantiate or retrieve an existing bean. General syntax is **<jsp:useBean id="beanName" class="className" scope="scopeValue" />**. Attribute **id** contains the name of the bean instance (reference variable) to be used. **class** contains the full class name of the bean. **scope** defines the scope in which the bean is available. It can be one of the following:

- **page (default)**: The bean is available only within the current page.
- **request**: The bean is available for the duration of the HTTP request.
- **session**: The bean is available for the duration of the HTTP session.
- **application**: The bean is available for the entire web application.

<jsp:setProperty>

The **<jsp:setProperty>** action is used to set the properties of a bean. It allows setting values in a bean based on user input (form data) or explicitly defined values. The general syntax is:

<jsp:setProperty name="beanName" property="propertyName" value="value" />

Attribute name contains the name (id) of the bean, property contains the name of the bean property to set. The value * can be used to automatically set all matching properties from request parameters. Value should be the value to be assigned to the property. The following example demonstrates setting a specific property:

<jsp:setProperty name="user" property="username"

value="john_doe" />

For automatically setting properties from request parameters the statement will be:

<jsp:setProperty name="user" property="*" />

In this example, all request parameters are matched with the bean properties and automatically set.

<jsp:getProperty>

The **<jsp:getProperty>** action retrieves the value of a bean's property and writes it to the output (typically HTML). The general syntax is:

<jsp:getProperty name="beanName" property="propertyName" />

Attribute name contains the name (id) of the bean and property defines the property of the bean to retrieve. For example:

** Username: <jsp:getProperty name="user" property="username" /> **

In this example, the value of the username property of the user bean is retrieved and displayed.

Here is how you can use the **UserBean** in a JSP page:

```
<%@ page language="java" contentType="text/
html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
<head>
    <title>JavaBean Example</title>
</head>
<body>

<jsp:useBean id="user" class="com.example.UserBean" scope="sessio
>

<!-- Setting properties of the bean -->
<jsp:setProperty name="user" property="name" value="John Doe" /
>
<jsp:setProperty name="user" property="email" value="john@examp
>
<!-- Displaying properties of the bean -->
<p>User Name: <jsp:getProperty name="user" property="name" /
```

```
    ></p>
15. <p>User Email: <jsp:getProperty name="user" property="email" />
    ></p>
16. </body>
17. </html>
```

The following table summarizes different action tags with examples:

	Description	
`<jsp:useBean id="user" class="com.example.User" />`		
`<jsp:setProperty name="user" property="*" />`		
`<jsp:getProperty name="user" property="username" />`		
`<jsp:include page="header.jsp" />`		
`<jsp:forward page="login.jsp" />`		
`<jsp:plugin type="applet" code="com.example.MyApplet" />`		
`<jsp:param name="id" value="123" />`		

***Table 13.2**: JSP action tags*

JSP implicit objects

JSP implicit objects are pre-defined objects available in every JSP page that provide convenient access to various data and functionalities related to the current request, response, session, and application. These objects simplify the development process by allowing developers to interact with key elements of a web application without explicitly declaring them. There are nine built-in implicit objects in JSP, each serving a specific purpose, request, response, out, session, application, config, page, pageContext, exception (in error pages). Let us discuss the common ones in detail.

request

In JSP, the request implicit object is a pre-defined object that represents the HTTP request received by the server from the client. It is an instance of the **javax.servlet.http.HttpServletRequest** interface and provides methods to obtain information about the request, such as request parameters, attributes, headers, and more.

- **Type**: **javax.servlet.http.HttpServletRequest**
- **Purpose**: Represents the client's request to the server. It allows access to request parameters, headers, attributes, and more.

Common methods:

- **getParameter(String name)**: Retrieves the value of a request parameter.
- **getAttribute(String name)**: Returns an attribute value associated with the request.
- **getHeader(String name)**: Retrieves the value of a specified request header.

Example:

```
<% String username = request.getParameter("username"); %>
```

response

In JSP, the response implicit object is a predefined object that represents the HTTP response sent by the server to the client. It is an instance of the **javax.servlet.http.HttpServletResponse** interface and provides methods for controlling and customizing the response, such as setting headers, cookies, and content types.

- **Type**: **javax.servlet.http.HttpServletResponse**
- **Purpose**: Represents the server's response to the client. It allows control over the response, such as setting content type, headers, and status codes.

Common methods:

- **setContentType(String type)**: Sets the MIME type of response.
- **sendRedirect(String location)**: Sends a temporary redirect response to the client.

Example:

```
<% response.setContentType("text/
html;charset=UTF-8"); %>
```

out

In JSP, the out implicit object is an instance of the **javax.servlet.jsp.JspWriter** class. It is

used to send output content to the client (typically a web browser). The out object is commonly used to print HTML content or dynamic data in the response.

- **Type**: **javax.servlet.jsp.JspWriter**
- **Purpose**: Used to send output to the client. It provides methods to write data directly to the response.

Common methods:

- **print(String s)**: Writes the string to the output.
- **println(String s)**: Writes the string followed by a new line.

Example:

```
1. <% out.println("Welcome to JSP!"); %>
```

session

In JSP, the session implicit object represents the HTTP session associated with a user. It is an instance of the **javax.servlet.http.HttpSession** interface and provides a way to maintain state across multiple requests from the same client.

- **Type**: **javax.servlet.http.HttpSession**
- **Purpose**: Represents the session for the current user. It allows storing and retrieving user-specific data across multiple requests.

Common methods:

- **getAttribute(String name)**: Retrieves an attribute from the session.
- **setAttribute(String name, Object value)**: Stores an

attribute in the session.

Example:

```
1. <% session.setAttribute("user", username); %>
```

application

In JSP, the application implicit object represents the servlet context for the entire web application. It is an instance of the **javax.servlet.ServletContext** interface and is used to share information across all servlets and JSP pages in the application. The application object has the widest scope and remains available as long as the web application is running.

- **Type**: **javax.servlet.ServletContext**
- **Purpose**: Represents the entire web application. It allows access to application-wide parameters and resources.

Common methods:

- **getAttribute(String name)**: Retrieves an application-wide attribute.
- **setAttribute(String name, Object value)**: Stores an attribute in the application context.

Example:

```
1. <
   % application.setAttribute("appName", "My Web Application"); %>
```

config

In JSP, the config implicit object provides access to the servlet configuration for the JSP page. It is an instance of the **javax.servlet.ServletConfig** interface and is used to retrieve initialization parameters and configuration information for the JSP page.

- **Type**: **javax.servlet.ServletConfig**
- **Purpose**: Provides configuration information for the JSP page. It allows access to initialization parameters defined in the **web.xml** file.

Common methods:

- **getInitParameter(String name)**: Retrieves an initialization

parameter.

Example:

```
1. <
   % String dbUrl = config.getInitParameter("databaseUrl"); %>
```

Integration of JSP with servlet

The integration of JSP and servlets is fundamental in Java web application development, as JSP and servlets complement each other. They are both part of the Java EE (now Jakarta EE) specification, with servlets being used to handle request processing and JSP being used for the presentation layer. This separation of concerns enables developers to build maintainable and scalable web applications.

Servlets are responsible for business logic, processing user input, handling forms, managing sessions, and interacting with databases. On the other hand, JSP focuses on presentation logic and is used for rendering the UI based on the data processed by servlets. By integrating Servlets and JSP, you can follow the **Model-View-Controller (MVC)** design pattern, where:

- **Model**: Represents the data and business logic, often handled by JavaBeans, POJOs, or database-related classes.
- **View**: The JSP pages, which display data to the user.
- **Controller**: The servlets act as intermediaries, processing requests, invoking business logic, and forwarding responses to JSP pages.

JSP and servlets integration can be used in scenarios where user requests and data processing can be handled in servlets and forwarded to JSP for Display (View). Let us walk through an example where a user submits a form via a JSP page, a servlet processes the data, and then forwards it back to another JSP for displaying results.

The following listing shows a JSP page where the user submits input data using an HTML form. This form sends the data to a servlet for processing:

```
1. <!-- index.jsp -->
2. <html>
```

```
<head><title>User Form</title></head>
<body>
    <h2>User Information Form</h2>
    <form action="UserServlet" method="post">
        Name: <input type="text" name="username" /><br/>
        Age: <input type="text" name="age" /><br/>
        <input type="submit" value="Submit" />
    </form>
</body>
</html>
```

The form's **action="UserServlet"** means the form data will be sent to a servlet named **UserServlet**. The **method="post"** attribute defines that the form uses an HTTP POST request. The following code implements the servlet (processing the request). The servlet processes the form data and then forwards it to a JSP page for rendering the results.

```
// UserServlet.java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class UserServlet extends HttpServlet {

    protected void doPost(HttpServletRequest request, HttpServletRespon
            throws ServletException, IOException {
        // Retrieving form data
        String username = request.getParameter("username");
        String age = request.getParameter("age");

        // Processing the data (e.g., simple validation)
        if (username != null && !username.isEmpty() && age !
= null && !age.isEmpty()) {
            request.setAttribute("userName", username);
            request.setAttribute("userAge", age);

            // Forwarding the request to a JSP page
            RequestDispatcher dispatcher = request.getRequestDispatcher(
            dispatcher.forward(request, response);
```

```
        } else {
            // Redirecting to an error page or back to the form
            response.sendRedirect("error.jsp");
        }
    }
}
```

doPost() method is called because the form uses the **POST** method. It retrieves form data using **request.getParameter("paramName")**. The servlet can perform any business logic, such as validating the input or interacting with the database. The servlet uses the **RequestDispatcher** to forward the processed data to the **display.jsp** page. It adds data to the request scope using **request.setAttribute()** so that the JSP page can access it. After the servlet processes the request, it forwards the data to this JSP page for display.

```
<!-- display.jsp -->
<html>
<head><title>Display User Information</title></head>
<body>
    <h2>User Details</h2>
    <!-- display.jsp -->
    <p>Name: <%= request.getAttribute("userName") %></p>
    <p>Age: <%= request.getAttribute("userAge") %></p>
</body>
</html>
```

The JSP page retrieves the attributes set by the servlet (**userName** and **userAge**) using **Expression Language (EL)** syntax: **${userName}** and **${userAge}**.

The JSP renders this information to the browser as part of the response.

Sometimes, one servlet may need to forward a request to another servlet before reaching a JSP. This can be done using **RequestDispatcher**. For example, **Servlet1** forwards the request to **Servlet2**:

```
// Forwarding from one servlet to another servlet
RequestDispatcher dispatcher =
request.getRequestDispatcher("SecondServlet");
```

dispatcher.forward(request, response);

JSP Standard Tag Libraries

JSP Standard Tag Library (JSTL) is a powerful library that provides a set of ready-to-use tags for common tasks in JSP development. JSTL simplifies the process of adding functionality such as iteration, conditionals, internationalization, database access, and XML manipulation to JSP pages. It reduces the need for Java code (scriptlets) within JSP pages, promoting cleaner and more readable code by using standardized tags. Simplifies common tasks such as looping, condition checking, and string manipulation. JSTL is divided into various function-specific tag libraries, each serving different purposes. They are Core Tags (c), Formatting Tags (fmt), SQL Tags (sql), XML Tags (x), Functions Tags (fn). Each library is included in the JSP page using the appropriate **taglib** directive.

The core tag library provides basic functionalities such as iteration, conditional logic, variable manipulation, and flow control.

Taglib directive used for core tags will be as follows:

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

Common JSTL Core Tags are listed below:

	Description	
<c:out value="${name}" />	Outputs the value of an expression or variable, optionally escaping HTML/XML special characters.	
<c:if test="${age > 18}">Adult</c:if>	Conditionally includes content based on a test expression.	
<c:choose>	Used as a switch statement with nested <c:when> and <c:otherwise> for multiple conditions.	
<c:when test="${status == 'active'}">	Subtag of <c:choose>, represents a case in a multiple condition test.	
<c:otherwise>	Subtag of <c:choose>, provides the default case when no other condition is true.	
<c:forEach var="item" items="${itemList}">	Iterates over a collection of items (lists, arrays, or maps).	
<c:set var="user" value="John Doe" />	Sets the value of a variable or JavaBean's scoped variable.	
<c:remove var="user" />	Removes a scoped variable from the session's scope.	
<c:redirect url="/success.jsp" />	Redirects to the "success.jsp" page.	
<c:catch>	Catches and handles exceptions that occur in the JSP page.	

***Table 13.3**: Common JSTL core tags*

The following example demonstrates the usage of core tags:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/
core" prefix="c" %>

<!-- Conditionally display content -->
<c:if test="${user.age >= 18}">
    <p>Welcome, adult user!</p>
</c:if>

<!-- Loop through a list -->
<ul>
    <c:forEach var="item" items="${shoppingCart}">
        <li>${item.name} - ${item.price}</li>
    </c:forEach>
</ul>

<!-- Set a variable and output it -->
<c:set var="total" value="100" />
<p>Total Price: <c:out value="${total}" /></p>
```

The formatting tag library helps with formatting numbers, dates, and currencies, as well as handling internationalization (i18n) by setting locales and loading resource bundles.

Taglib directive for formatting tag library is as follows:

<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>

Common JSTL formatting tags are listed below:

	Description	
fmt:formatNumber	**Formats a number value, e.g., <fmt:formatNumber value="1234.56" type="currency" />**	
fmt:formatDate	**Formats a date value, e.g., <fmt:formatDate value="${now}" pattern="yyyy-MM-dd" />**	
fmt:parseNumber	**Parses a number value, e.g., <fmt:parseNumber value="1,234.56" />**	
fmt:setLocale	**Sets the locale for formatting, e.g., <fmt:setLocale value="en_US" />**	
fmt:setBundle	**Sets the resource bundle for internationalization.**	
fmt:bundle	**Loads a resource bundle for messages, e.g., <fmt:bundle basename="messages">...</fmt:bundle>**	
fmt:message	**Retrieves a message from a resource bundle, e.g., <fmt:message key="welcome.message" />**	

***Table 13.4**: Common JSTL formatting tags*

Following example demonstrates formatting numbers and dates:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/
fmt" prefix="fmt" %>

```

```
3. <p>Current Date: <fmt:formatDate value="${currentDate}" pattern=
   dd/yyyy" /></p>
4. <p>Price: <fmt:formatNumber value="${price}" type="currency" /
   ></p>
```

JSP session management

In JSP, session management allows you to store and retrieve user-specific data between multiple requests. The data is stored on the server side and associated with a unique session ID, which is typically passed back and forth between the client and the server via cookies or URL rewriting.

HTTP sessions

JSP supports HTTP sessions by default. A session is created when a user accesses the application for the first time, and the server generates a unique session ID, which is sent to the user as a cookie or encoded in the URL. The session can store data using key-value pairs, which can be retrieved across multiple requests. A session is automatically created when a JSP page is accessed for the first time unless it is explicitly disabled. You can also explicitly create one. Data can be stored in a **Session** by using **session.setAttribute()**. **session.getAttribute()** is used to retrieve stored data. You can manually end a session by using **session.invalidate()**. The below given program demonstrates the session management:

```
1. <%@ page language="java" session="true" %>
2. <%
3.   // Storing data in session
4.   String username = request.getParameter("username");
5.   session.setAttribute("userName", username);
6. %>
7.
8. <!-- Displaying session data -->
9. <p>Welcome, <%= session.getAttribute("userName") %>!
   </p>
```

In this example, the session stores the user's name when it is retrieved from the form input (**username**). The user's name is stored using **session.setAttribute("userName", username)**, which can be accessed in later requests. The session data is displayed using

session.getAttribute().

Methods used in HTTP sessions:

- **session.setAttribute(String name, Object value)**: Stores an object in the session.
- **session.getAttribute(String name)**: Retrieves the object associated with the given name.
- **session.removeAttribute(String name)**: Removes the object associated with the given name from the session.
- **session.invalidate()**: Destroys the session and all data associated with it.

Cookies in JSP

A cookie is a small piece of data stored by the web browser on the client side. You can use cookies in JSP for session management by storing session data in the form of cookies, which will be sent along with every request. In order to use cookies in JSP, create a **Cookie** using new **Cookie(name, value)**. Use **response.addCookie()** to send the cookie to the browser. To retrieve cookies from the browser **request.getCookies()** is used. Cookies can have a specified lifetime (expiry time), after which they will be deleted.

Below given example demonstrates using Cookies:

```
<%@ page language="java" %>
<%
    // Creating a cookie
    String username = request.getParameter("username");
    Cookie userCookie = new Cookie("userName", username);

    // Set cookie expiration time to 24 hours
    userCookie.setMaxAge(24 * 60 * 60);

    // Adding cookie to the response
    response.addCookie(userCookie);
%>

<!-- Retrieving cookie -->
<%
    Cookie[] cookies = request.getCookies();
```

```
    String userName = null;

    if (cookies != null) {
        for (Cookie cookie : cookies) {
            if (cookie.getName().equals("userName")) {
                userName = cookie.getValue();
                break;
            }
        }
    }
%>

<p>Welcome, <%= userName !
= null ? userName : «Guest" %>!</p>
```

In the above example, a cookie is created using new **Cookie("userName", username)** and added to the response using **response.addCookie(userCookie)**. The browser stores the cookie and sends it back with every subsequent request. On subsequent requests, the cookie is retrieved using **request.getCookies()**, and the value is extracted.

URL rewriting in JSP

In URL rewriting, the session ID is appended to the URL as a query string parameter, and this helps maintain the session even if cookies are disabled on the client-side. URL rewriting is particularly useful when cookies are not supported or not enabled by the user. Below given example demonstrates URL rewriting:

```
<%@ page language="java" session="true" %>
<a href="nextPage.jsp?<
%= response.encodeURL("sessionid=" + session.getId()) %>">Go to
a>
```

The **response.encodeURL()** method ensures that the session ID is appended to the URL if cookies are not enabled. This allows the server to recognize the user's session across pages. The session ID is appended to the URL as a query string.

Hidden form fields in JSP

Hidden form fields can also be used for session management. These

fields are not visible to the user but are passed between server and client when the form is submitted. The session data is stored as part of the form and sent to the server with every form submission. The following example demonstrates hidden form fields:

```
<%@ page language="java" session="true" %>
<form action="nextPage.jsp" method="POST">
    <input type="hidden" name="sessionid" value="<
%= session.getId() %>" />
    <input type="submit" value="Go to Next Page" />
</form>
```

A hidden field is used to store the session ID. When the form is submitted, the session ID is sent to the server, and the session can be tracked.

You can configure the session timeout duration in the **web.xml** file. If the session is inactive for this duration, it will expire automatically:

```
<session-config>
    <session-timeout>30</session-timeout>  <!-- In minutes -->
</session-config>
```

Let us see one example application of session handling in JSP. This application is implemented with a **login.jsp** for login and session initialization, a **loginServlet.java** for processing login and storing session data, **welcome.jsp** for displaying session data and **logout.jsp** for invalidating the session.

login.jsp:

```
<%@ page language="java" session="true" %>
<html>
<head><title>Login</title></head>
<body>
    <form action="loginServlet" method="POST">
        Username: <input type="text" name="username" /
><br/>
        Password: <input type="password" name="password" /
><br/>
        <input type="submit" value="Login" />
    </form>
</body>
</html>
```

loginServlet.java:

```
1. import java.io.*;
2. import javax.servlet.*;
3. import javax.servlet.http.*;
4.
5. public class loginServlet extends HttpServlet {
6.     protected void doPost(HttpServletRequest request, HttpServletRespor
7.             throws ServletException, IOException {
8.         // Retrieve form data
9.         String username = request.getParameter("username");
10.        String password = request.getParameter("password");
11.
12.        // Dummy login check
13.        if ("admin".equals(username) && "admin".equals(password)) {
14.            // Store username in session
15.            HttpSession session = request.getSession();
16.            session.setAttribute("userName", username);
17.
18.            // Redirect to a welcome page
19.            response.sendRedirect("welcome.jsp");
20.        } else {
21.            // Redirect to login failure page
22.            response.sendRedirect("loginFailure.jsp");
23.        }
24.    }
25. }
```

welcome.jsp:

```
1. <%@ page language="java" session="true" %>
2. <html>
3. <head><title>Welcome</title></head>
4. <body>
5.     <h1>Welcome, <
   %= session.getAttribute("userName") %>!</h1>
6.     <form action="logout.jsp" method="POST">
7.         <input type="submit" value="Logout" />
8.     </form>
9. </body>
10. </html>
```

logout.jsp:

```
<%@ page language="java" session="true" %>
<%
   session.invalidate();
%>
<html>
<head><title>Logout</title></head>
<body>
   <h1>You have successfully logged out.</h1>
</body>
</html>
```

Conclusion

In conclusion, JSP serves as a powerful tool for creating dynamic, interactive web applications by integrating seamlessly with Java Servlets. Through this chapter, we have explored JSP's lifecycle, architecture, and various elements like directives, actions, and JSTL. The integration of JSP and servlets offers a robust framework for separating presentation from business logic, adhering to the MVC design pattern. This results in web applications that are not only scalable and maintainable but also easier to develop. In the next chapter, we will delve deeper into advanced topics, further enhancing our knowledge of web development with JSP. We will learn about dynamic web content with JSP.

Exercises

1. Build a simple login form using JSP. The form should have two input fields for username and password. Validate the input and display an appropriate message based on the input.
2. Develop a JSP page that demonstrates session tracking. Create a shopping cart system where users can add items to their cart. Use JSP sessions to track the items the user adds. Display the cart contents on a separate page, ensuring the session persists between requests.
3. Create a JavaBean that holds user data (name, email). Develop a controller servlet that processes a form submission and sets the JavaBean properties. Use a JSP page to display

the data from the JavaBean in a user-friendly format.

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 14
Modern Web Development with JSF

Introduction

JavaServer Faces (JSF) is a popular framework for building user interfaces for Java-based web applications. As part of the Java EE ecosystem, JSF simplifies the development of web applications by providing a component-based approach for managing user interface elements. This framework facilitates the creation of reusable UI components and allows developers to define the application's behavior through backing beans, event handling, and model-driven architecture. Its support for rich user interfaces and seamless integration with other Java technologies makes JSF a powerful tool for web development. This chapter discusses the structure, architecture, and processing lifecycle of JSF, positioning it within the broader landscape of web frameworks.

Structure

The chapter covers the following topics:

- JavaServer Faces framework

- Comparison with other web frameworks
- The architecture of JSF web applications
- Building a JSF application

Objectives

The objective of this chapter is to introduce the JSF framework, focusing on its key components and architectural design. By the end of the chapter, readers will understand how JSF compares to other web frameworks, its request processing lifecycle, and how to build a simple JSF web application. The chapter will also explore the role of configuration files such as **faces-config.xml** and **web.xml**, which are integral to the organization of web resources in JSF applications. The goal is to equip readers with foundational knowledge to create and manage JSF-based web applications effectively.

JavaServer Faces framework

JSF is a Java-based web application framework designed to simplify the development of user interfaces for web applications. It is part of the Java **Enterprise Edition** (**EE**) platform and provides a component-based approach where developers can build reusable UI components that manage their behavior, rendering, and data binding. Key features of the JSF Framework are as follows:

- **Component-based architecture**: JSF is based on reusable UI components, such as buttons, forms, and tables, that can be used to create dynamic, server-side web interfaces. Each component has its own lifecycle, and JSF manages how it interacts with the underlying data model and user inputs.
- **Managed beans**: Managed beans are Java classes that act as controllers, handling the logic behind UI components. JSF automatically manages the lifecycle of these beans, and they are often used to process input data, handle user actions, and maintain the application state.
- **Event-driven**: JSF applications are event-driven, meaning that user actions (like button clicks) trigger events. These events are handled on the server side, allowing for complex

interactions without the need for extensive JavaScript.

- **UI component tree:** JSF organizes components in a hierarchical tree structure called the component tree. The framework processes this tree in phases, including rendering, restoring the view, applying user inputs, validating data, and updating the model.
- **Rich customizable UI**: JSF supports both standard and custom UI components. Developers can use built-in components provided by the JSF library, or they can create their own to meet specific requirements. Moreover, third-party component libraries like PrimeFaces can be integrated to add richer functionalities and themes.
- **Separation of concerns**: JSF follows the **Model-View-Controller** (**MVC**) design pattern. The model handles the application data, the view is responsible for rendering the UI, and the controller manages the interactions between the two. This clear separation of concerns makes JSF applications easier to develop, maintain, and test.
- T**ight Integration with Java EE**: JSF is closely integrated with Java EE technologies such as **Enterprise JavaBeans** (**EJB**), **Java Persistence API** (**JPA**), and **Contexts and Dependency Injection** (**CDI**). This allows developers to leverage the full capabilities of the Java EE platform to build enterprise-level web applications.
- **Extensible and customizable**: Developers can extend JSF by creating custom components, validators, converters, and event listeners. The framework is designed to be flexible and adaptable, allowing for customization to suit a wide range of project needs.

JSF applications typically use XHTML pages as views. These views define the UI components and specify how they bind to data in the managed beans. When a user interacts with the application, JSF follows a well-defined request processing lifecycle, which includes phases like restoring the view, applying request values, processing validations, updating the model, and rendering the response.

JSF also leverages the concept of navigation rules to determine the next view to display after an action is performed. These rules can be

defined declaratively in configuration files (**faces-config.xml**) or using annotations.

Comparison with other web frameworks

JSF, when compared with other popular web frameworks, several key differences emerge in terms of architecture, development approach, ease of use, flexibility, and target audience. Here is a detailed comparison of JSF with other common web frameworks like Spring MVC, Struts, and Angular.

JSF vs. Spring MVC

JSF and Spring MVC are two popular Java-based web frameworks that follow the MVC architecture, yet they differ in their design philosophies, approaches, and use cases.

JSF is a component-based framework maintained by **Oracle**. It is designed to simplify the development of user interfaces for Java EE applications. JSF provides reusable UI components, event-driven development, and strong integration with the Java EE platform. It operates at a higher level of abstraction, allowing developers to focus on building views with minimal code for controlling the behavior.

On the other hand, Spring MVC is part of the Spring Framework and is action-based, meaning it focuses more on controlling the flow of the application through controllers that map user requests to actions. It is lightweight and flexible, making it popular in modern web development. Spring MVC's architecture allows you to use RESTful services, view technologies, and it provides tight integration with the Spring ecosystem.

Differences between JSF and Spring MVC:

	Spring MVC	
Component-based UI framework		
Uses JSP, Facelets, or other templating engines		
Supports two-way data binding		
Strong integration with Java EE		
Requires understanding of Spring architecture		
Highly flexible, suitable for RESTful modern web apps		
Easy integration (e.g., with Angular, React)		
Modern web applications with RESTful services, microservices		

Table 14.1: Differences between JSF and Spring MVC

JSF is ideal for applications where the UI complexity is high, and a component-based architecture is preferred, especially in enterprise environments using Java EE. In contrast, Spring MVC provides more flexibility, is well-suited for RESTful services, and can be easily integrated with modern web technologies. The choice between JSF and Spring MVC depends largely on project requirements, the team's familiarity with the framework, and the desired level of control over the web application development process.

JSF vs. Struts

Apache Struts is another Java-based web frameworks that follow the MVC design pattern and is widely used in Java web development. Struts is well-suited for applications that emphasize controller logic and action handling. It offers more flexibility in terms of view rendering and is easier to extend and customize due to its interceptor-based architecture. Struts is a good choice for developers who want more control over the request-response cycle and who are comfortable building UIs using templating engines like JSP.

The following table shows the differences between JSF and Struts:

	Struts	
Component-based, UI framework	Action-based, MVC framework	
Managed beans	Action classes	
Facelets, JSP, or other templating engines	JSP, Freemarker, or other templating engines	
Supports AJAX out of the box	Requires plugins or custom code for AJAX	
Stateful by default, state is managed automatically	Stateless by default, state must be managed explicitly	
Tightly integrated with Java EE technologies like CDI, JPA	Can be integrated with Java EE technologies like Spring, Hibernate	
Easier for non-developers to create UIs	Requires more coding for UI development	
Built-in validation and conversion	Requires custom validation and conversion code	
Limited flexibility in customizing UI components	More flexible in customizing UI components	
Steeper learning curve	Easier learning curve	
Suitable for complex, enterprise-level applications	Suitable for simple to medium-sized applications	

Table 14.2: JSF vs Struts

JSF vs. Angular

JSF and client-side frameworks like Angular, React, and Vue.js represent two distinct approaches to web development. JSF is a server-side, component-based Java framework, while Angular and

similar client-side frameworks are focused on building dynamic, highly interactive user interfaces that operate mostly on the client-side.

JSF is part of the Java EE ecosystem, designed to simplify the creation of web applications by providing a component-based approach with reusable UI elements. JSF applications typically rely on server-side rendering, where the server generates and sends fully rendered HTML to the client.

Angular, React, and Vue.js are client-side frameworks that rely on the **Single Page Application (SPA)** model, where much of the UI logic is handled by JavaScript in the browser. These frameworks allow for more dynamic, interactive user interfaces by managing client-side rendering and using AJAX for data updates without full page reloads. The following table shows the key differences between JSF vs. Angular (and other client-side frameworks):

	Angular (React, Vue.js)	
Server-side rendering	Client-side rendering	
Uses HTML templates with JSF tags	Uses HTML templates with JavaScript/TypeScript	
Server-side state management (managed beans)	Client-side state management (Redux or Vuex)	
Uses partial page updates via AJAX	Uses full page updates or AJAX	
Typically uses RESTful APIs or GraphQL	Typically uses RESTful APIs or GraphQL	
Built-in support for rich UI components	Requires third-party libraries for UI components	
Slower for highly interactive UIs	Faster for highly interactive UIs	
Suitable for enterprise-scale applications	Suitable for enterprise-scale applications	
Steeper learning curve	Steeper learning curve	
Server-side logic is hidden from the client	Client-side logic is visible to the client	
Enterprise apps with complex UIs	SPAs, mobile-friendly applications	

***Table 14.3**: JSF vs. Angular (and other client-side frameworks)*

JSF and Angular (or React/Vue.js) represent two distinct paradigms in web development. JSF is server-side, component-based, and deeply integrated into the Java EE ecosystem, making it ideal for enterprise applications with complex UI requirements. On the other hand, Angular, React, and Vue.js are client-side frameworks designed for building dynamic, highly interactive UIs, and they excel in developing modern SPAs and mobile-friendly applications.

The choice between them depends on the project requirements. For UI-heavy, interactive applications, Angular and similar frameworks offer a better experience. For Java EE-based enterprise solutions, JSF might still be the preferred choice due to its close integration

with Java and strong server-side capabilities.

The following table summarizes the key differences between different frameworks:

***Table 14.4**: Comparison between different frameworks*

The architecture of JSF web applications

The architecture of a JSF web application follows the MVC design pattern, where JSF acts as a component-based framework that handles **user interface** (**UI**) components on the server. It provides a robust mechanism for building complex, server-side UI applications, managing the component lifecycle, and handling user events.

Model-View-Controller architecture

In MVC architecture, model is the business logic and data management part of the application. In a JSF application, the model is typically represented by JavaBeans (managed beans or backing beans) that contain data and business logic.

View, the presentation layer is responsible for displaying the user interface. In JSF, the view is defined using Facelets pages (usually written in XHTML) and JSF UI components.

The controller handles user input, processes actions, and coordinates between the model and the view. In JSF, the controller functionality is managed internally by the JSF framework, and actions are often handled by methods in backing beans.

Key components of JSF architecture are as follows:

JSF UI components

JSF is a component-based framework where user interfaces are constructed using a variety of reusable UI components. These

components include buttons, forms, tables, and input fields, which are declared in Facelets pages and processed by the JSF framework. JSF Tags represent UI components in the Facelets pages, such as <h:form>, <h:inputText>, <h:commandButton>.

Each component tag maps to a corresponding Java class, such as UIForm or UIInput. These classes manage the component's behavior and state.

Managed beans

Managed beans or backing beans are the backbone of the business logic and act as the Model in JSF architecture. They handle user interactions and business logic and store the state of the components. Managed beans are often annotated with @ManagedBean or @Named (from CDI), and their scope is defined using annotations like @RequestScoped, @SessionScoped, or @ViewScoped.

These beans contain action methods that are invoked when the user interacts with components like buttons or links. The methods can return navigation outcomes.

Facelets view

Facelets is the default view technology in JSF, used to create the UI. These XHTML pages define the layout and components of the UI using JSF tags and expressions. The structure of the page is defined using HTML and JSF tags, e.g., <h:form>, <h:outputText>, <h:dataTable>. JSF uses **Expression Language** (**EL**) to bind UI components to managed beans, e.g., #{bean.property}.

FacesServlet

The FacesServlet is the central controller in JSF, responsible for processing incoming HTTP requests and invoking the JSF lifecycle. It acts as the entry point for all JSF requests and delegates tasks like rendering the view, handling events, and updating the model. The FacesServlet is mapped to handle requests ending with **.xhtml** or **.jsf** in the **web.xml** configuration file.

Navigation

JSF provides a navigation mechanism that determines which view should be displayed after a user action is executed. Navigation rules can be configured in the faces-config.xml file or using annotations directly in the managed beans. Static navigation is defined in **faces-config.xml**.

Dynamic navigation is handled by managed beans. Managed beans can return outcomes as strings to trigger page navigation dynamically.

Validation and conversion

JSF includes a built-in mechanism for validation and data conversion. Validators and converters ensure that input values conform to expected formats or constraints. Converters convert data from one type to another, e.g., String to Date, using @FacesConverter or built-in converters like f:convertDateTime. Validators validate user input using @FacesValidator or built-in validators like f:validateLength.

Event handling

Event handling in the JSF architecture is centered around user interactions with UI components and their lifecycle. JSF uses a JavaBeans-based event model, where events generated by UI components are handled by managed beans or custom listeners. JSF provides a robust event-driven model for handling various user actions (e.g., button clicks, value changes). Managed beans can define listeners (e.g., ActionListener, ValueChangeListener) to respond to events generated by UI components.

Managed bean scopes

The scope of a managed bean defines its lifecycle within the application:

- **Request scope (@RequestScoped)**: The bean is created and destroyed with each request.
- **Session scope (@SessionScoped)**: The bean exists for the duration of a user session.
- **View scope (@ViewScoped)**: The bean persists across multiple requests on the same view.

- **Application scope (@ApplicationScoped)**: The bean is shared across the entire application. The following diagram summarizes the architecture of JSF web applications:

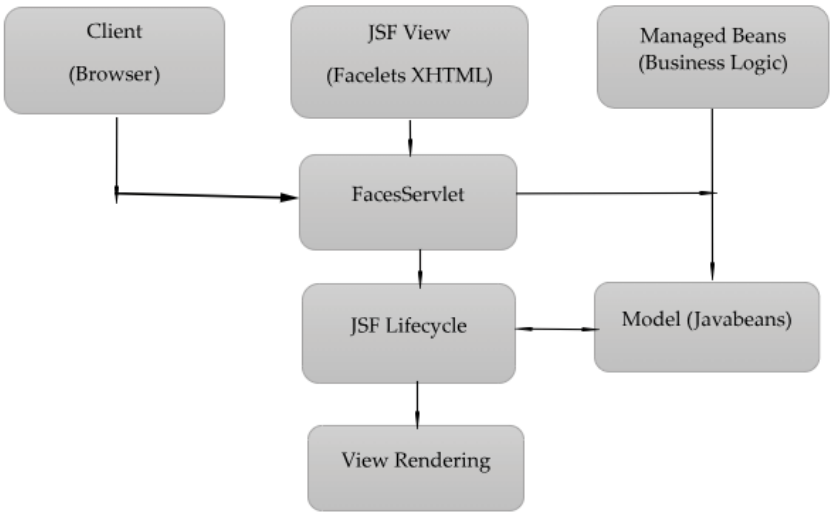


***Figure 14.1**: JSF architecture*

Lifecycle phases of JSF request processing

The JSF request processing lifecycle is a well-defined sequence of phases that an HTTP request goes through when interacting with a JSF application. The lifecycle is responsible for handling user input, converting and validating it, updating the model, invoking application logic, and finally rendering the response back to the client. Understanding these phases is crucial for controlling and customizing the behavior of a JSF application. There are six main phases in the JSF lifecycle:

- **Restore view phase**: The restore view phase is the first step in the JSF lifecycle. In this phase, the framework either creates a new view for the request or restores the view from a previous request. A view in JSF represents the UI component tree of the page, which includes all JSF components (forms, buttons, input fields, etc.). If it is the first time the page is being accessed, JSF creates an empty component tree. If the request is a postback (the user is

submitting data from a form), JSF restores the component tree from the previous request (saved on the server or client). During this phase, JSF sets up the UI component tree for the view but does not process any input or handle user actions yet.

- **Apply request values phase**: In the apply request values phase, JSF processes the incoming HTTP request parameters (usually form data) and assigns these values to the corresponding UI components. The values submitted by the user (e.g., text input, checkboxes, dropdown selections) are applied to the components. If a component has the **immediate="true"** attribute set, its values are processed immediately in this phase, bypassing the later phases (like validation and model update). If needed, JSF applies converters to the input data to transform it into the correct type (e.g., converting a string to a date). In this phase, the values from the HTTP request are captured by the corresponding UI components but are not yet validated or applied to the model.
- **Process validations phase**: During the process validations phase, JSF validates the data entered by the user according to the constraints defined on the UI components (such as length, format, and required fields). JSF has built-in validators for common constraints, such as f:validateLength, f:validateRegex, and f:validateRequired. You can create custom validators that implement the Validator interface. If validation fails, error messages are added to the FacesContext, and the lifecycle skips to the render response phase, so the page can be re-rendered with validation error messages. This phase checks the validity of the user input. If validation fails, the user is informed, and the invalid data is not applied to the model.
- **Update model values phase**: In the update model values phase, JSF updates the values of the managed beans (the model) with the validated input from the UI components. The UI components are bound to properties of managed beans using EL, e.g., #{userBean.username}. In this phase, the component values are assigned to the corresponding

bean properties. If any conversion or validation errors occurred in previous phases, this phase is skipped. This is where the validated and converted user input is applied to the model (managed beans). If any issues occurred earlier, this phase is not executed.

- **Invoke application phase**: The Invoke Application phase is where business logic is executed. This phase processes the actions initiated by the user, such as form submissions, button clicks, and link navigation. Managed beans often contain methods (e.g., #{userBean.login}) that are invoked in this phase. These methods contain the business logic of the application, such as logging in, updating records, or processing payments. Based on the result of the action method, JSF decides which view to display next. This is determined either by the return value of the method (e.g., "success" leads to a success page) or by navigation rules defined in faces-config.xml. In this phase, JSF executes any user-triggered actions and determines the next view or page to display.
- **Render response phase**: The Render Response phase is the final step in the JSF lifecycle. In this phase, JSF generates the HTML for the UI components and sends it back to the client (typically a web browser). For an initial request, this phase builds and renders the UI component tree as an HTML page. For a postback, if there are any validation errors, the page is re-rendered with error messages. Otherwise, the updated component tree is rendered. The rendered HTML is sent back to the client, where it is displayed in the browser. In this phase, JSF converts the component tree back into HTML (or another suitable format) and sends it to the client. If errors occur during validation, they are displayed to the user. The following figure shows various stages of JSF lifecycle:

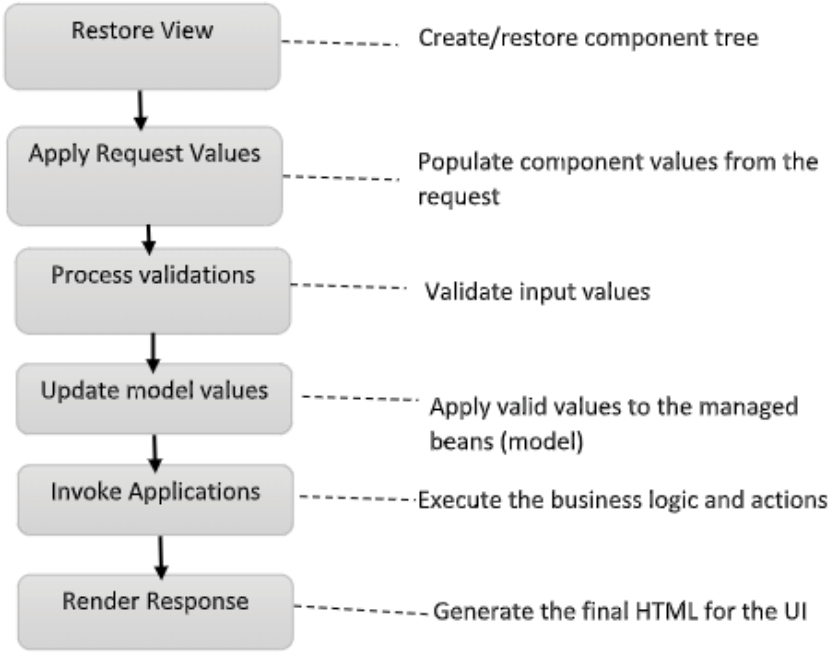


***Figure 14.2**: JSF lifecycle*

The JSF lifecycle can be short-circuited under certain conditions:

- **Validation errors**: If validation or conversion errors occur, JSF jumps directly to the render response phase to display the error messages and does not proceed to the model update or application invocation phases.
- **Immediate attribute**: Components with **immediate="true"** are processed earlier in the lifecycle (during the apply request values phase), potentially skipping validations and model updates for other components.

Customizing the lifecycle

JSF allows developers to customize the lifecycle through various mechanisms:

- **Phase listeners**: You can register listeners for specific phases of the lifecycle to execute custom logic before or after each phase.
- **Immediate attribute**: You can alter the flow of the lifecycle

for specific components by setting the **immediate="true"** attribute.

Building a JSF application

To develop a JSF application, you will need a few essential tools and dependencies. To set up a JSF application, install the tools listed below:

- **Java Development Kit (JDK)**: JSF requires Java. Download the latest JDK version (17 or higher is recommended) from the Oracle website or OpenJDK.
- **Apache Maven**: Maven is a popular build automation tool for Java projects. You can download it from Maven's official site. To install Maven on Windows, add the Maven bin directory to your system's PATH after extraction. To install Maven on macOS/Linux, you can use package managers like **brew install maven (macOS)** or **sudo apt install maven (Linux)**. Verify the installation by running **mvn -version** in the terminal, which should show the installed version.
- **Integrated development environment (IDE)**: Use an IDE like Eclipse or IntelliJ IDEA, which simplifies project management and offers extensive support for JSF.
- **Eclipse IDE**: Download Eclipse from **https://www.eclipse.org/downloads/**. The Eclipse IDE for Java EE Developers package includes JSF support.
- **IntelliJ IDEA**: Download IntelliJ IDEA (Community or Ultimate) from **https://www.jetbrains.com/idea/download/**.
- **Web server (Apache Tomcat or GlassFish)**: JSF applications require a server to run. Apache Tomcat and GlassFish are popular servers for Java EE applications.
- **Tomcat**: Download from **https://tomcat.apache.org/** and follow the setup instructions.
- **GlassFish**: Download from **https://javaee.github.io/glassfish/**, and follow the setup instructions.

Creating a Maven project with JSF dependencies

The following are the steps to create a Maven project with JSF dependencies:

1. Open your IDE and create a new Maven project:

   Open Eclipse or IntelliJ IDEA.

   Go to **File | New | Maven Project.**

   When prompted to select an archetype, choose the maven-archetype-webapp archetype. This archetype sets up a basic structure for a web application.

1. Configure project details:

   Provide a Group ID (e.g., **com.example.jsf**) and an Artifact ID (e.g., **JSFUserRegistration**).

   Set the version (default is 1.0-SNAPSHOT).

   Choose the location to save the project, then click **Finish**.

1. Add JSF Dependencies to **pom.xml**:

   Once the project is created, open the **pom.xml** file (found in the root directory of your project) and add the following JSF dependencies:

```
<!-- pom.xml -->
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/
POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example.jsf</groupId>
    <artifactId>JSFUserRegistration</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>

    <dependencies>
      <!-- JSF API and Implementation -->
      <dependency>
          <groupId>javax.faces</groupId>
          <artifactId>javax.faces-api</artifactId>
          <version>2.3</version>
          <scope>provided</scope>
      </dependency>
```

```
        <dependency>
            <groupId>org.glassfish</groupId>
            <artifactId>javax.faces</artifactId>
            <version>2.3.9</version>
        </dependency>
    </dependencies>

    <build>
        <finalName>JSFUserRegistration</finalName>
    </build>
</project>
```

In this configuration:

The **javax.faces-api** dependency provides the core JSF API, and **javax.faces** contains the reference implementation. Adjust versions as per your project requirements (version 2.3 is commonly used and compatible with many servers).

The scope for **javax.faces-api** is set to provided to indicate that the server (e.g., Tomcat) will supply this dependency at runtime.

1. Configure **web.xml** for JSF:

   In a Maven web application, the web.xml file is located in **src/main/webapp/WEB-INF**. This configuration file maps the JSF servlet to process **.xhtml** files.

   Open or create **web.xml** in the WEB-INF directory.

   Add the following configuration:

```
<!-- web.xml -->
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
version="3.1">
    <display-name>JSFUserRegistration</display-name>

    <!-- JSF Servlet Configuration -->
    <servlet>
        <servlet-name>FacesServlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</
servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
```

```
    <servlet-name>FacesServlet</servlet-name>
    <url-pattern>*.xhtml</url-pattern>
  </servlet-mapping>

  <welcome-file-list>
    <welcome-file>index.xhtml</welcome-file>
  </welcome-file-list>
</web-app>
```

The **FacesServlet** is the core servlet of JSF, responsible for managing the lifecycle of JSF pages.

The **<url-pattern>*.xhtml</url-pattern>** mapping tells the server that all **.xhtml** files should be processed by JSF.

index.xhtml is set as the welcome file, which means it will be the default page loaded when the application starts.

1. Create the directory structure and basic files:

   To complete the setup, create the directory structure and basic pages for your JSF application.

   In the **src/main/webapp** directory, create a new file named **index.xhtml**:

```
<!-- index.xhtml -->
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
<h:head>
  <title>JSF Application</title>
</h:head>
<h:body>
  <h1>Welcome to JSF!</h1>
</h:body>
</html>
```

Run the application on your web server:

- **Eclipse IDE**: Right-click on the project, choose **Run As | Run on Server**, and select the server you installed.
- **IntelliJ IDEA**: Deploy the application on Tomcat or GlassFish via Run Configurations.

Navigate to **http://localhost:8080/JSFUserRegistration/** in your

browser. You should see the "**Welcome to JSF!**" message displayed on the page.

Structure of a typical JSF web application

The structure of a typical **JavaServer Faces** (**JSF**) web application generally follows the standard Java web application architecture. This structure organizes components, resources, and configurations in a way that enables smooth development, maintenance, and deployment. Below is an outline of the key directories and files typically found in a JSF web application.

Project root directory

At the top level of a Maven project, you will have the **pom.xml** file along with folders like **src/**. This structure can vary slightly if using Gradle or another build tool.

pom.xml: Maven configuration file where dependencies, build configurations, and plugins are defined. Dependencies for JSF libraries (like **javax.faces-api**) are listed here.

Source code directory (src/main/java)

All Java source files, such as Managed Beans and backend classes, are located here:

- **Managed beans**: Java classes annotated with @ManagedBean, @RequestScoped, @SessionScoped, etc., which handle user data, form submissions, and business logic.
- **Service classes (optional)**: Logic processing, often organized separately from managed beans to separate concerns.
- **Model classes (optional)**: Java classes representing data entities. They are useful for applications involving complex data structures.

Example directory structure:

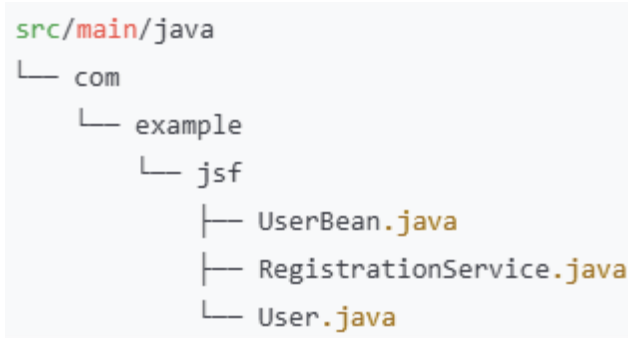


***Figure 14.3**: Source code directory structure*

Web resources directory (src/main/webapp)

Contains HTML and JSF **.xhtml** pages, CSS, JavaScript, and configuration files:

- **index.xhtml**: Entry point for the application (configured as a welcome file in **web.xml**).
- **JSF Pages (.xhtml)**: These are Facelet pages that use JSF tags for component rendering. They interact with Managed Beans to present and process user data.
- **Resources Folder (src/main/webapp/resources)**: Stores static resources like CSS, JavaScript, and images in subdirectories (e.g., **css/**, **js/**, **images/**).
- WEB-INF:

  The **WEB-INF/** directory is a special folder in a Java web application that serves as a secure location for configuration files, libraries, and server-side resources that should not be directly accessible by users via a browser. Typical contents of **WEB-INF/** are listed below.
- **web.xml**: Configuration file for web applications, including JSF servlet mapping, security settings, and welcome files.
- **faces-config.xml (optional)**: JSF configuration file, used to configure beans, navigation rules, and custom components. While no longer strictly required for managed bean definitions, it is still useful for navigation and additional settings.

Example directory structure:

```
src/main/webapp
├── index.xhtml
├── resources
│   ├── css
│   │   └── styles.css
│   ├── js
│   │   └── script.js
│   └── images
│       └── logo.png
└── WEB-INF
    ├── web.xml
    └── faces-config.xml (optional)
```

***Figure 14.4**: Web Resources directory structure*

Configuration and properties

In addition to **web.xml** and **faces-config.xml**, the resources folder often contains application properties for internationalization (i18n) and custom configurations:

- **messages.properties**: Properties file for localizing messages, labels, and other text used in the application.
- **Custom configurations**: Files for any specific application settings (like database connections) can also be stored here or directly under **src/main/resources**.

Project structure summary

Here is how a complete JSF project directory might look:

***Figure 14.5**: Project structure*

This directory structure keeps your JSF project organized, separating the business logic (Java classes), view templates (**.xhtml** files), configuration files, and resources. It is a typical layout for most Maven-based JSF applications and provides a scalable foundation for more complex projects.

Configuration files

In JSF architecture, configuration files play a crucial role in defining the behavior and structure of the application. Two primary configuration files are commonly used:

- **faces-config.xml**: The faces-config.xml file is the main configuration file for a JSF application, defining managed beans, navigation rules, and other global settings. Managed beans can be declared here (though annotations are more commonly used today). Static navigation rules between pages are configured in this file.
- **web.xml**: This is the standard deployment descriptor for Java web applications. In JSF, it is used to configure the following.
    - **Servlet mappings**: Maps JSF requests to the FacesServlet (typically through **.xhtml** or **.jsf** file extensions).
      Context parameters, such as for handling error pages or configuring the behavior of the application.
    - **Session timeout**: Defines session behavior, such as timeout periods.
    - **Error handling**: Configures custom error pages for exceptions or status codes. The FacesServlet mapping.

Let us build a simple **User Registration** application with JSF. This application will include:

- A form for user input (username and password)
- A managed bean to process the data
- Displaying the entered data on a success page
- Create a Maven project and set up JSF dependencies

  Use the following dependencies for a Maven project:

```
<!-- pom.xml -->
<dependencies>
  <!-- JSF API and Implementation -->
  <dependency>
    <groupId>javax.faces</groupId>
    <artifactId>javax.faces-api</artifactId>
```

```
            <version>2.3</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.glassfish</groupId>
            <artifactId>javax.faces</artifactId>
            <version>2.3.9</version>
        </dependency>
    </dependencies>
```

- Configure **web.xml** for JSF
- Set up the **web.xml** file to configure JSF mappings:

```
<!-- web.xml -->
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
version="3.1">
    <display-name>JSFUserRegistration</display-name>
    <!-- JSF Servlet Configuration -->
    <servlet>
        <servlet-name>FacesServlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</
servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>FacesServlet</servlet-name>
        <url-pattern>*.xhtml</url-pattern>
    </servlet-mapping>
    <welcome-file-list>
        <welcome-file>index.xhtml</welcome-file>
    </welcome-file-list>
</web-app>
```

- Create a managed bean for User Registration
- Managed beans handle form data and business logic in JSF

```
// src/main/java/com/example/jsf/UserBean.java
package com.example.jsf;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean(name = "userBean")
```

```
@SessionScoped
public class UserBean {

    private String username;
    private String password;

    // Getters and Setters
    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    // Action method to navigate to the success page
    public String register() {
        return "success";
    }
}
```

- Create JSF pages for user registration:
    - **index.xhtml**: The index page will be used as registration form

      This page contains a simple form for entering the username and password:

```
<!-- src/main/webapp/index.xhtml -->
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
<h:head>
    <title>User Registration</title>
```

```
    </h:head>
    <h:body>
        <h2>User Registration</h2>

        <h:form>
            <h:outputLabel for="username" value="Username: " />
            <h:inputText id="username"
value="#{userBean.username}" required="true"/>
            <br/>

            <h:outputLabel for="password" value="Password: " />
            <h:inputSecret id="password"
value="#{userBean.password}" required="true"/>
            <br/>

            <h:commandButton value="Register"
action="#{userBean.register}" />
        </h:form>
    </h:body>
    </html>
```

- **success.xhtml**: Success page

  This page will display the registered user's details:

```
<!-- src/main/webapp/success.xhtml -->
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
<h:head>
    <title>Registration Success</title>
</h:head>
<h:body>
    <h2>Registration Successful!</h2>

    <h:outputText value="Welcome, #{userBean.username}!
Your registration is successful." />
</h:body>
</html>
```

- Run and test the application
- Build the application in your IDE or with Maven

- Deploy it on a web server, like Tomcat
- Navigate to **http://localhost:8080/JSFUserRegistration/** to access the registration form
- When you submit the form with a username and password, it will redirect to the **success.xhtml** page and display the entered username

In the above application, UserBean is a managed bean with **@ManagedBean** annotation that holds the user's data and includes an action method (register) to handle the form submission.

index.xhtml is the main form page where users input their data. JSF expressions (e.g., **#{userBean.username}**) are used to bind form data to the UserBean. **success.xhtml** is a confirmation page that displays the username entered by the user.

Conclusion

JavaServer Faces offers a robust and flexible framework for developing user interface components in Java web applications. Its ability to integrate seamlessly with other Java EE technologies, coupled with its component-based architecture, makes JSF a reliable choice for developers looking to build scalable, maintainable applications. Understanding its request processing lifecycle, configuration, and application structure allows developers to optimize the user experience and maintain complex applications with ease. This chapter serves as a gateway to mastering JSF, laying the groundwork for advanced topics in Java-based web development.

In the next chapter, we will learn about web frameworks.

Exercises

1. Create a simple "Hello World" JSF application. Create a managed bean named HelloBean. Add a string property message with the value "Hello, World!". Create an XHTML page with a JSF form that displays the message.
2. Build a basic calculator for addition, subtraction, multiplication, and division. Create a managed bean called CalculatorBean with properties num1, num2, and result.

Create methods to perform each calculation. In the XHTML page, add input fields for num1 and num2, and buttons to trigger each operation. Display the result below the buttons.

3. Create a login form that validates the username and password. Create a managed bean named LoginBean with username and password properties. Add a method validateLogin() to check if the username is "admin" and password is "password". Display a success message if the credentials are correct; otherwise, show an error message. Use JSF input fields and message components to display validation errors.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 15
Java Beans and Web Frameworks

Introduction

This chapter explores Java's powerful web development frameworks, providing a structured approach to building scalable, maintainable, and modular web applications. We begin with an introduction to JavaBeans, a concept previously discussed in the JSP chapter. By using a basic JavaBean class, we explored how to create objects that hold data attributes and provide standardized methods (getters and setters) to access and modify them. Now, in this chapter, we will delve deeper into JavaBeans, examining their architecture, properties, and conventions in detail, and explore how they fit into the larger Java ecosystem for building modular, reusable code components. The chapter then dives into three of the most widely used Java frameworks: Spring MVC, Struts and Hibernate. We will explore each framework's architecture, key components, lifecycle, and distinct advantages. Through a comparative study, this chapter will guide readers in understanding the strengths and ideal use cases of each framework, enabling them to make informed decisions in framework selection based on project

requirements.

Structure

The chapter covers the following topics:

- Introduction to JavaBeans
- Java Web Frameworks
- Hibernate framework
- Comparison between the frameworks

Objectives

This chapter aims to provide a comprehensive overview of popular Java web development frameworks, focusing on the role of JavaBeans and how these frameworks facilitate efficient, scalable, and modular web application development. This chapter will explore JavaBeans and its integration within frameworks like Spring MVC, Hibernate, and Struts. By examining the architecture, core components, and lifecycle of each framework, readers will gain insights into their distinct advantages and best use cases, empowering them to choose and implement the appropriate framework for diverse project requirements.

Introduction to JavaBeans

JavaBeans are reusable software components in the Java programming language that follow a set of conventions, allowing them to be easily used, configured, and manipulated within various Java applications, especially in visual development environments. The concept of JavaBeans was introduced to enhance code reusability and component-based software development, promoting modularity in Java applications.

At their core, JavaBeans are Java classes that adhere to certain standards, primarily focusing on properties, events, and persistence. These standards make JavaBeans particularly useful for creating user interface elements, custom objects, and other components that can be configured and integrated seamlessly into larger applications. Key features of JavaBeans are listed below:

- **Properties**: JavaBeans expose their attributes, or properties, through getter and setter methods. For instance, if a bean has a property called name, it would have **getName()** and **setName()** methods to retrieve and modify its value. Properties can be of various types, including simple, indexed, or bound, enabling versatile data encapsulation.
- **Events**: JavaBeans support an event model similar to the listener model in Java, where beans can generate events, and other objects can listen for and respond to them. This capability is essential in GUI applications and interactive software, where user actions trigger responses in the application.
- **Persistence**: A key characteristic of JavaBeans is that they are designed to be serializable, meaning their state can be saved and restored. This enables beans to persist across sessions and be used in distributed applications.
- **Introspection**: JavaBeans support introspection, which allows development tools to analyze the properties, methods, and events of a bean dynamically. Through introspection, developers can use beans without delving into the underlying code, making them highly adaptable to various visual development environments.

Benefits of using JavaBeans

The following are the benefits of using JavaBeans:

- **Reusability**: JavaBeans can be reused across multiple applications and projects without modification.
- **Encapsulation**: Beans encapsulate their data and behavior, exposing only what's necessary through public methods.
- **Interoperability**: Beans are highly compatible with a variety of frameworks and development tools, making them flexible components in different environments.

The common applications of JavaBeans:

JavaBeans are widely used in Java Enterprise applications, desktop applications, and frameworks that rely on reusable, modular components. They form the basis for many enterprise frameworks,

and JavaBeans architecture also underpins Java EE components such as **Enterprise JavaBeans (EJBs)** and managed beans in **JavaServer Faces (JSF)**.

Creating a simple JavaBean: PersonBean

Let us create a simple JavaBean named PersonBean. This bean will have two properties: name (a String) and age (an int). It will include getter and setter methods for these properties, a no-argument constructor, and will implement the Serializable interface to allow the bean to be saved and retrieved if needed.

```
import java.io.Serializable;
// The PersonBean class represents a simple JavaBean with name and age properties.
public class PersonBean implements Serializable {

    private static final long serialVersionUID = 1L;

    // Private properties (fields)
    private String name;
    private int age;

    // No-argument constructor
    public PersonBean() {
    }

    // Getter and setter methods for name
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    // Getter and setter methods for age
    public int getAge() {
        return age;
    }

    public void setAge(int age) {
```

```
        this.age = age;
    }

    // A method to display person information
    public String getPersonInfo() {
        return "Name: " + name + ", Age: " + age;
    }
}
```

PersonBean implements Serializable, which is a requirement for a JavaBean if it needs to be saved and retrieved later. The name and age fields are private to encapsulate the data. These fields are only accessible through public getter and setter methods. The default constructor (without arguments) allows frameworks and tools to instantiate PersonBean without needing parameters. The **getName()** and **setName()** methods access and modify the name property, while **getAge()** and **setAge()** handle the age property. These follow the standard JavaBean naming conventions. Custom method (**getPersonInfo**) provides a formatted output of the person's information, which could be useful in the application to display or log a person's details. However, since it deals with fields like name and age that could potentially include personally identifiable information (PII), ensuring the safe handling of this data is essential. To ensure PII is not stored or retrieved inadvertently, particularly in the context of Java beans, sessions, and serialization, you can mark sensitive fields as transient. In Java, fields marked as transient are excluded from the default serialization process. This is crucial for sensitive PII data that should not be serialized when the object is written to a file or transmitted over a network. Below are the recommendations and changes to align with the best practices for handling PII:

```
import java.io.Serializable;
// The PersonBean class represents a secure JavaBean with name and age pro
public class PersonBean implements Serializable {
    private static final long serialVersionUID = 1L;
    // Private properties (fields)
    private transient String name;
  // Marked as transient to avoid serialization of PII
    private transient int age;
   // Marked as transient to avoid serialization of PII
```

```
    // No-argument constructor
    public PersonBean() {
    }
    // Getter and setter methods for name
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    // Getter and setter methods for age
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    // A method to display person information
    public String getPersonInfo() {
        return "Name: " + (name != null ? name : "Not Available") +
               ", Age: " + (age > 0 ? age : "Not Available");
    }
}
```

By marking the name and age fields as transient, the fields are excluded from serialization. This ensures PII is not logged or stored unintentionally during serialization. When JavaBeans are part of HTTP sessions, the application server (e.g., Tomcat) may serialize session objects to disk during shutdown or failover. Marking sensitive fields as transient ensures PII is not serialized into these session files.

Using PersonBean in an application

Here is an example of how to use PersonBean in an application. We will create an instance, set its properties, retrieve them, and display the information:

```
public class MainApp {
    public static void main(String[] args) {

        // Create an instance of PersonBean
```

```
        PersonBean person = new PersonBean();

        // Set properties using setter methods
        person.setName("Alice");
        person.setAge(30);

        // Get and display individual properties
        System.out.println("Name: " + person.getName());
        System.out.println("Age: " + person.getAge());

        // Display all information using getPersonInfo method
        System.out.println(person.getPersonInfo());
    }
}
```

We create a **PersonBean** object using the default constructor. We use the **setName()** and **setAge()** methods to assign values to the name and age properties. To retrieve individual property values, we use **getName()** and **getAge()**, which print **"Alice"** and **"30"** respectively. Finally, we use the getPersonInfo() method, which outputs a formatted string of the person's details as:

Name: Alice, Age: 30

Although the use of a default constructor, may introduce vulnerabilities by allowing the creation of incomplete or invalid bean objects. To address this, we can introduce a constructor with mandatory fields (name and age) to ensure beans are always instantiated with valid data. This prevents the accidental creation of partially initialized objects and enhances data integrity.

The concept of JavaBean was discussed precisely in *Chapter 13, Dynamic Web Content with JSP*. It was introduced as a core component in building dynamic, reusable Java components for web applications. We saw how JavaBeans act as containers for data, encapsulating properties and allowing seamless interaction with JSP through simple tags.

Java Web Frameworks

Java Web Frameworks are software platforms that provide tools, libraries, and best practices for developing and deploying web

applications in Java. These frameworks streamline the development process by offering pre-built components and structures, allowing developers to focus more on business logic and user experience rather than repetitive tasks and low-level details. Java Web Frameworks simplify web development by handling common tasks such as database interaction, form handling, session management, and security.

They promote code reuse, maintainability, and scalability by organizing code in a structured way. These frameworks enforce a clear separation of concerns (often through MVC architecture), where the Model handles data, the View manages the user interface, and the Controller processes user input, improving readability and making collaboration easier.

Popular Java Web Frameworks are Spring Framework, Struts, etc. The purpose of Java Web Frameworks is:

- **Standardization**: They provide standard methods for handling common tasks, such as request routing, view rendering, and data persistence.
- **Efficiency**: By streamlining routine processes, frameworks reduce the need for boilerplate code, allowing developers to focus on the unique aspects of an application.
- **Scalability**: Frameworks include built-in support for scaling web applications, allowing them to handle large amounts of traffic and data.
- **Security**: Most frameworks integrate security mechanisms to protect applications from common vulnerabilities like SQL injection, cross-site scripting, and request forgery.

Spring MVC Framework

The Spring MVC Framework is a part of the larger Spring Framework ecosystem, primarily designed to simplify the creation of web applications. It follows the MVC pattern, which separates the application's data (Model), user interface (View), and control flow (Controller) to promote a more organized and manageable code structure. Let us discuss components, architecture, and key features of Spring MVC in detail.

Spring MVC architecture

The Spring Framework is designed to simplify Java development by providing a comprehensive, modular, and layered architecture. This architecture is composed of multiple modules organized into core components that cater to different aspects of an enterprise application. The core layers of Spring Architecture are listed below:

- **Core container**: This is the foundational layer of the Spring Framework, providing the fundamental functionality of Spring.

  It consists of the *Core* and *Beans* modules, which provide **Inversion of Control** (**IoC**) and **dependency injection** (**DI**) capabilities, allowing developers to manage application configurations and dependencies in a flexible way.

  The Context module builds on the Core and Beans modules to provide a means to access objects in a framework-style manner. It includes a large set of services, such as event propagation, resource loading, and access to various contexts (e.g., web contexts for web applications).

  The EL module provides a powerful expression language for querying and manipulating an object graph at runtime.

- **Aspect-oriented programming (AOP)**: This layer provides support for aspect-oriented programming, which enables separating cross-cutting concerns (like logging, security, and transaction management) from the main business logic.

  The AOP module allows developers to define method interceptors and pointcuts, so that they can apply these cross-cutting concerns declaratively.

  The Aspects module allows integration with AspectJ, a powerful aspect-oriented programming framework.

- **Data access/integration**: This layer is designed to streamline data access within an application and offers integration with various persistence APIs. It includes:

  - **JDBC**: JDBC Simplifies database connectivity by eliminating boilerplate code, managing exceptions, and handling transactions.

- **Object-relational mapping (ORM)**: ORM integrates with ORM frameworks like Hibernate and JPA, making it easier to work with persistent data.
- **Transaction management**: It provides a consistent, flexible approach to declarative transaction management across various data management technologies.
- **Messaging**: It supports integration with message brokers through the Spring Messaging module, which includes support for messaging protocols and WebSocket communication.

- **Web**: This layer provides support for creating web applications. Web module contains basic web-oriented integration features, such as multipart file upload, using Servlet listeners to initialize the IoC container, and web application context.
  - Web-Servlet (Spring MVC) module contains Spring's MVC framework, allowing for web application development with a clear separation of concerns.
  - Web-Socket module supports WebSocket-based, real-time, two-way communication.
  - Web-Portlet module offers an MVC implementation for use in a portlet environment, making it suitable for handling web requests in a portal environment.
- **Test**: The test module supports the testing of Spring components with JUnit or TestNG frameworks. It provides mock objects and support for dependency injection, enabling unit testing of Spring services, DAOs, and controllers in isolation.

Spring MVC module

The Spring MVC module is part of the Web layer in the Spring architecture and is one of the most popular modules for building web applications. The key components of Spring MVC are:

- **DispatcherServlet**: At the heart of the Spring MVC framework is the DispatcherServlet, a front controller that

routes incoming HTTP requests to the appropriate handler methods. The DispatcherServlet acts as a bridge between the client and the backend of the application. It is the front controller that intercepts incoming HTTP requests and delegates them to the appropriate handler.

- **Controller**: The controller is a class that handles and processes specific requests from the client, interacts with the model to gather data, and then selects a view to render the response.
- **Model**: This represents the application's data and business logic. Controllers can add attributes to the model, making data available for views. It can include JavaBeans, database access, and any other logic for retrieving and storing information.
- **View**: The view layer defines how data is presented to the user. It defines the UI representation and can be implemented with JSP, Thymeleaf, or other view technologies. Views receive data from the model to render the response.
- **Handler mapping**: Maps URLs to specific controller methods. For example, RequestMappingHandlerMapping links request paths to @RequestMapping annotated methods.
- **View resolver**: Resolves the view name returned by the controller to a specific view implementation, such as a JSP or Thymeleaf template.

In a Spring MVC request flow, a request is received by DispatcherServlet. HandlerMapping identifies the appropriate controller. The controller processes the request, interacts with the model, and returns a view name. ViewResolver resolves the view name, and the view is rendered to the user.

Spring bean lifecycle

Spring beans are objects that are instantiated, assembled, and managed by the Spring IoC (Inversion of Control) container. The lifecycle of a bean includes several stages:

- **Instantiation**: The Spring container instantiates the bean

using the class specified in the configuration (either XML, annotations, or Java configuration).

- **Property population (dependency injection)**: After instantiation, the container performs dependency injection by setting the bean's properties. This includes injecting any dependencies the bean might need, such as other beans, configuration properties, or resources.
- **Aware interfaces (optional)**: If a bean implements any of the "Aware" interfaces (e.g., BeanNameAware, BeanFactoryAware, ApplicationContextAware), the container will call the corresponding methods. These methods provide the bean with access to certain Spring-related resources like its own name (setBeanName), the BeanFactory instance (setBeanFactory), or the ApplicationContext (setApplicationContext).
- **Bean initialization (custom initialization methods)**: Spring allows beans to specify custom initialization methods through the @PostConstruct annotation or by implementing the InitializingBean interface with an afterPropertiesSet method. Additionally, you can specify an init-method in XML configuration or use the @Bean annotation with initMethod in Java configuration. This custom initialization logic is executed after dependency injection and aware methods are complete.
- **Post-initialization with BeanPostProcessors**: If there are any BeanPostProcessor implementations in the container, they can intercept beans after initialization. There are two main methods involved:
    - **postProcessBeforeInitialization**: This method is called before any initialization callbacks.
    - **postProcessAfterInitialization**: This method is called after any initialization callbacks.

  These methods are often used for custom modifications to beans or for implementing proxy objects, such as for aspect-oriented programming (AOP) proxies.
- **Bean ready for use**: After all initialization steps are

complete, the bean is fully ready for use in the application. The application can use the bean as needed until the context is shut down.

- **Bean destruction (custom destruction methods)**: When the application context is closed, Spring initiates the bean destruction process. Any custom destruction logic specified by the bean (using @PreDestroy, DisposableBean interface with a destroy method, destroy-method in XML, or **@Bean(destroyMethod="...")** in Java configuration) is executed. This phase is where you can release resources, close connections, or perform any cleanup necessary before the bean is removed from memory.

The following are the key interfaces and annotations:

- **BeanNameAware, BeanFactoryAware, ApplicationContextAware**: Access to bean metadata and container resources.
- **InitializingBean (afterPropertiesSet), @PostConstruct, init-method**: Custom initialization methods.
- **BeanPostProcessor**: Custom logic before/after initialization.
- **DisposableBean (destroy), @PreDestroy, destroy-method**: Custom destruction methods.

Spring API

The Spring API offers a wide range of classes, interfaces, and utilities across its different modules. Here are some of the core components:

- **ApplicationContext**: It extends BeanFactory and provides advanced features like event propagation, declarative mechanisms, and layered resource handling. Common implementations include ClassPathXmlApplicationContext and AnnotationConfigApplicationContext.
- **BeanFactory**: It is the root interface for accessing Spring's IoC container. It manages the creation and configuration of beans, handling DI.
- **BeanPostProcessor**: It is an interface for custom modifications to beans. It allows logic to be injected before

and after bean initialization.

- **BeanFactoryPostProcessor:** It allows bean definitions to be modified before the container instantiates them.
- **resourceloader**: It is an interface for loading resources (like files, URLs, and classpath resources). Allows Spring to abstract file handling for different environments.
- **Environment**: It provides access to environment properties and profiles, allowing configuration to change based on environments like development, testing, or production.
- **ApplicationEventPublisher**: It is a utility for publishing events in the Spring application context, enabling loosely coupled components.
- **PlatformTransactionManager**: It is the base interface for transaction management. It has implementations for various transaction models, including DataSourceTransactionManager for JDBC, JpaTransactionManager for JPA, and HibernateTransactionManager for Hibernate.

Key annotations in the Spring API

Spring offers a powerful set of annotations that make it easy to configure applications, manage dependencies, and streamline the overall development process. Here is a list of some of the most commonly used annotations in the Spring API, organized by functionality:

- **@Autowired**: Enables automatic dependency injection by type.
- **@Qualifier**: Used alongside @Autowired to resolve ambiguity when multiple beans of the same type exist.
- **@Component, @Service, @Repository, and @Controller**: Stereotype annotations that mark classes as Spring beans. They indicate the role of each component in the application and can be scanned automatically for dependency injection.
- **@Configuration and @Bean**: Indicate that a class provides Spring bean definitions. @Configuration classes hold @Bean methods, each returning an instance of a bean managed by

Spring.

- **@Value**: Injects values from properties files, environment variables, or expressions directly into fields, methods, or constructors.
- **@Transactional**: Manages transaction boundaries, helping with commit/rollback handling in data access layers.
- **@EventListener**: Registers a method to listen for specific events published within the application context.

The Spring Framework offers flexibility and modularity for developing Java applications, whether they are simple web applications, complex enterprise systems, or RESTful APIs. With its powerful IoC and DI, MVC, AOP, and transaction management features, Spring provides a complete development ecosystem for various application architectures.

Request handling workflow in Spring MVC

Here is how a typical request is handled:

1. The client sends a request to the application's URL.
2. The DispatcherServlet intercepts the request.
3. The DispatcherServlet consults the HandlerMapping to identify the appropriate controller and method.
4. The identified controller method is invoked, often interacting with the model (e.g., service layer) to retrieve or process data.
5. The controller method returns a ModelAndView object or view name with data, which the ViewResolver uses to render the final view.
6. The ViewResolver identifies and loads the appropriate view template, which is then populated with the model data and sent back to the client as the response.

Form handling and data binding

Spring MVC provides a robust way to handle form data. The framework uses data binding to automatically map HTTP request parameters to Java objects (POJOs), allowing form data to be

validated and processed as Java objects.

- **Form submission**: Forms are designed with fields corresponding to properties in a model object.
- **Binding and validation**: Spring MVC binds form data to the model object and can validate it with annotations like @NotNull, @Size, etc., using the @Valid annotation.
- **Error handling**: If errors occur, they can be collected in BindingResult and displayed to the user in the view.

Exception handling in Spring MVC

Spring MVC provides flexible exception handling mechanisms:

- **@ExceptionHandler**: Defined at the controller level to handle exceptions thrown by a specific controller.
- **@ControllerAdvice**: A global exception handler that can manage exceptions thrown by multiple controllers. It centralizes exception handling logic for the entire application.

RESTful web services in Spring MVC

Spring MVC can be easily configured to handle RESTful services by returning data directly (usually as JSON or XML) instead of views. Key annotations include:

- **@RestController**: Used to build REST APIs, returning JSON responses by default.
- **@RequestMapping (and related HTTP-specific annotations like @GetMapping, @PostMapping, etc.)**: To define endpoint paths and HTTP methods.
- **@RequestBody and @ResponseBody**: Allow for automatic conversion of JSON/XML request and response bodies to and from Java objects.

Advantages of using Spring MVC

The following are advantages of using Spring MVC:

- **Flexibility and extensibility**: Spring MVC integrates easily with various view technologies, templating engines, and libraries, allowing flexibility in design.

- **Decoupled components**: Clear separation of concerns between components (controller, model, view) supports maintainable, modular code.
- **Annotation-based configuration**: Reduces boilerplate configuration by allowing direct annotations within classes and methods.
- **Rich ecosystem**: Seamlessly integrates with other Spring modules (like Spring Boot, Spring Data, and Spring Security), which further simplifies development.
- **Testability**: Controllers and other components are easier to test, especially with Spring's support for dependency injection.

Spring Boot and Spring MVC

Spring Boot is commonly used to simplify Spring MVC configuration by providing auto-configuration and embedded servers. With Spring Boot:

- You can set up a Spring MVC application with minimal configuration.
- It includes an embedded Tomcat server, so you can run the application without needing a separate server.
- Spring Boot simplifies dependencies and packaging, ideal for quickly deploying Spring MVC applications.

Sample Spring MVC code

Here is a simple example of a Spring MVC controller handling a request:

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class GreetingController {

    @GetMapping("/greeting")
            public String greeting(@RequestParam(name = "name",
```

```
required = false, defaultValue = "World") String name, Model model) {
        model.addAttribute("name", name);
        return "greeting"; // returns the view name 'greeting'
    }
}
```

The **@Controller** annotation marks this class as a Spring MVC controller. **@GetMapping("/greeting")** maps HTTP GET requests to **/greeting** to this method. The method accepts a name parameter, which is added to the model and passed to the view (**greeting.jsp** or another view technology). Spring MVC is a powerful framework for developing dynamic, scalable web applications and, when used with Spring Boot, makes setup and development even more efficient.

Struts Framework

The Struts Framework is an open-source Java framework primarily used for building web applications based on the MVC design pattern. It provides a structured way to build large-scale applications by separating concerns: Model (business logic), View (presentation), and Controller (request handling). Initially developed by the Apache Software Foundation, Struts has evolved into Struts 2, a modern version incorporating significant improvements, such as better integration with other frameworks and an improved configuration model. Key features of Struts Framework are:

- **MVC architecture**: Provides a clean separation between business logic and user interface.
- **POJO-based actions**: Allows developers to work with plain old Java objects (POJOs) instead of implementing framework-specific interfaces.
- **Flexible interceptors**: Provides a mechanism to add functionality to specific points in the request lifecycle, such as authentication or logging.
- **Tag libraries**: Includes custom tags that simplify the creation of web pages, such as **s:form** and **s:textfield** tags.
- **Easy integration**: Integrates well with other Java frameworks, such as Spring and Hibernate.

Core components

The core components of the Struts Framework enable the MVC design pattern, making it easier to manage large applications by separating concerns. Here are the primary components of Struts:

- **ActionServlet**: The ActionServlet is the main controller of a Struts application. It intercepts all incoming requests, initializes resources, and delegates processing to the appropriate Action classes. ActionServlet reads configuration information from the struts-config.xml file, which defines mappings between URLs, actions, and views. This component is crucial for routing user requests to specific actions and managing the application's lifecycle.
- **ActionForm**: ActionForm is a JavaBean that represents the form data submitted by a user in the view (JSP). It acts as a bridge between the client-side form and the server-side logic. Each ActionForm captures and validates input data, and then makes it available to the Action class. Struts uses these forms to hold data between requests, reducing the need for additional form-handling code and helping with data validation.
- **Action**: The Action class contains business logic for handling specific requests. Each Action class represents a particular request or business action (e.g., submitting a form or querying a database). The Action class processes the data from the ActionForm, executes the required business logic, and returns an ActionForward to define the next step (e.g., a new JSP view). Action classes separate business logic from the view, providing a clean way to manage user interactions.
- **ActionMapping**: ActionMapping defines the configuration of each Action in the struts-config.xml file. It maps a specific URL pattern to an Action class and includes information about the ActionForm to use, input validation settings, and the associated ActionForward paths. This mapping simplifies routing and allows for easy management of URL-to-action associations within a centralized configuration.
- **ActionForward**: ActionForward defines the navigation path after an action is completed. It determines which view

(typically a JSP or another action) should display the result.

After the Action class finishes processing, it returns an ActionForward object specifying the next view or resource. ActionForward can be internal (pointing to a JSP within the application) or external (redirecting to an external URL).

- **struts-config.xml**: **struts-config.xml** is the central configuration file in Struts, where mappings between actions, forms, and views are specified. It contains ActionMapping definitions, references to ActionForm beans, global exception handling, and data source configurations. This file provides a comprehensive view of the application's navigation structure, making it easy to adjust mappings and settings without changing code.
- **Struts tag libraries**: Struts provides several custom tag libraries, which simplify form handling, logic processing, and data display in JSPs:
    - **HTML tags**: For creating HTML form elements (e.g., <**html:text**>, <**html:form**>) with built-in data binding.
    - **Bean tags**: For accessing JavaBeans and displaying bean properties.
    - **Logic tags**: For conditional logic (e.g., <**logic:equal**>, <**logic:iterate**>) within JSPs.

    These tags streamline JSP development by reducing embedded Java code, making views more maintainable and readable.

Configuration files

The following are the configuration files:

- **struts.xml**: The core configuration file for mapping actions, results, and interceptors. It defines the flow of the application, such as which actions to invoke and which results to display.
- **web.xml**: A standard Java web application descriptor that configures the application, including the ActionServlet and other resources.

Struts architecture

The Struts architecture follows the MVC pattern, with each component playing a specific role:

- **Model**: Represents the application's business logic, often implemented with Java classes or integrated with Hibernate for data access. The Model interacts with the database to retrieve and manipulate data.
- **View**: The presentation layer, often implemented with JSP pages and Struts Tag Libraries, displays data to the user and collects input. The view is only responsible for presenting data and does not contain any business logic.
- **Controller**: Manages the flow of the application, processing user requests, calling the appropriate Action class, and directing the response. The ActionServlet acts as the central controller, receiving all incoming requests.

Struts Request Flow

It centralizes request handling and response generation, promoting modularity and separation of concerns. The following is a detailed explanation of how a request flows through the Struts framework, from the client interaction to the server response:

- **Client request**: A user makes a request to the server.
- **Request handling**: The ActionServlet intercepts the request and consults struts.xml to determine which Action to invoke.
- **Action processing**: The Action class processes the request. It may use a model class to retrieve or update data as needed.
- **Result forwarding**: After processing, the Action returns a result, which is defined in struts.xml. This result specifies the view (e.g., a JSP page) to display to the user.
- **Response generation**: The specified view renders the response, which is sent back to the client.

This architecture ensures a clean separation between the layers, simplifying maintenance and enhancing reusability.

Interceptors

Interceptors are a powerful feature in Struts 2 that allow for common, reusable code to be executed before and after the Action processing. They help manage cross-cutting concerns, such as logging, validation, authentication, and transaction management. It facilitates both pre-processing and post-processing, allowing developers to implement reusable logic at different stages of the action execution cycle.

- **Pre-processing**: Interceptors can execute logic before the Action is invoked, such as validating user input or checking if a user is authenticated.
- **Post-processing**: They can also execute logic after the Action has completed, such as logging or cleaning up resources.

Core interceptors in Struts

Core interceptors in Struts2 are built-in interceptors that provide essential functionalities for handling common concerns such as validation, parameter management, and exception handling. These interceptors are part of the framework's defaultStack, which is applied to actions unless explicitly overridden. Key core interceptors are listed below.

- **ValidationInterceptor**: Validates input data before the action executes. It checks if the data in the form fields meets the specified criteria, ensuring data quality before proceeding.
- **ParamsInterceptor**: Maps request parameters to action properties, allowing automatic data binding from form fields to Action properties.
- **ModelDrivenInterceptor**: Helps populate the model object in an action. It simplifies the retrieval and management of model data by automatically injecting a model object.
- **LoggingInterceptor**: Logs information about each request, including parameters and results, for debugging and auditing purposes.
- **ExceptionInterceptor**: Manages exceptions thrown by actions, allowing specific handling of errors and providing

meaningful feedback to users.

- **WorkflowInterceptor**: Controls the flow of the application. If an action fails validation, it forwards the user to an error page; otherwise, it allows the action to execute.

Custom interceptors

Developers can also create custom interceptors by implementing the interceptor interface, defining **init()**, **intercept()**, and **destroy()** methods. Custom interceptors provide flexibility for unique application requirements, such as custom authorization checks or data pre-processing.

Interceptor Stack

In Struts, interceptors are grouped into stacks to organize their execution order. Some common stacks include:

- **Default stack**: Executes default interceptors like ParamsInterceptor, ValidationInterceptor, and others in a specific order.
- **Custom stacks**: Developers can create custom stacks by configuring interceptors in struts.xml to execute a specific sequence of interceptors for certain actions.

Example of interceptor configuration in struts.xml:

```
<package name="default" namespace="/" extends="struts-default">
    <interceptors>
        <interceptor name="log"
class="com.example.interceptors.LoggingInterceptor"/>
        <interceptor-stack name="customStack">
            <interceptor-ref name="log"/>
            <interceptor-ref name="validation"/>
        </interceptor-stack>
    </interceptors>

    <action name="login" class="com.example.LoginAction">
        <interceptor-ref name="customStack"/>
        <result name="success">/welcome.jsp</result>
        <result name="error">/error.jsp</result>
```

```
    </action>
</package>
```

In this example, the **customStack** includes a **LoggingInterceptor** and **ValidationInterceptor**. The login action uses this stack, ensuring that logging and validation are performed for every login request.

Validator framework

Struts includes a built-in validation framework, which allows developers to define validation rules in an XML file (**validation.xml**) or directly in the ActionForm classes. The Validator Framework simplifies form validation by managing error messages and validation logic outside of the main application code, reducing redundancy and improving maintainability.

Validators in Struts2 handle data validation to ensure user input meets specific criteria before it reaches the action logic. Struts2 offers both XML-based and annotation-based validation, allowing flexible configuration. Validators can be field validators or non-field validators. **Field validators** applied to specific fields within a form (e.g., required fields, email format, string length). **Non-field validators** perform validation across multiple fields or other custom logic, such as verifying that two fields match (e.g., password and confirm password).

Examples of different types of validators are given below:

- **Required field validator**: Ensures that a field is not left blank.

```
<field name="username">
    <field-validator type="requiredstring">
        <message>Username is required.</message>
    </field-validator>
</field>
```

- **Email validator**: Checks if a field contains a valid email format.

```
<field name="email">
    <field-validator type="email">
        <message>Invalid email address.</message>
```

```
    </field-validator>
</field>
```

- **Custom validator**: You can create custom validators for more complex validation logic.

  You can also use annotations for validation, which can make your code cleaner and more intuitive.

```
import com.opensymphony.xwork2.validator.annotations.*;

public class User {
    @RequiredStringValidator(message = "Username is
required.")
    private String username;

    @EmailValidator(message = "Invalid email address.")
    private String email;

    // Getters and setters
}
```

- **Configuring validators**: Validators can be defined in validation.xml files associated with actions, or in Java classes using annotations. For example, **LoginAction-validation.xml** would define validation rules for LoginAction.

Hibernate Framework

The Hibernate Framework is an open-source **object-relational mapping** (**ORM**) framework for Java. It simplifies the process of mapping Java objects to relational database tables and provides a powerful query language (HQL) for database operations. Hibernate abstracts complex SQL queries and manages database transactions, caching, and concurrency, making it easier to build database-driven applications. The key features of Hibernate are as follows:

- **Transparent persistence**: Hibernate manages the persistence of objects automatically, so developers do not need to write boilerplate code for CRUD operations.
- **Hibernate Query Language (HQL)**: HQL is an object-

oriented query language, similar to SQL but works with objects rather than tables and columns.

- **Lazy loading**: Hibernate supports lazy loading of objects, meaning that data is fetched only when needed, which helps optimize performance.
- **Caching**: Hibernate supports caching mechanisms like first-level and second-level caches to reduce database calls.
- **Automatic table generation**: Hibernate can generate the database tables automatically based on object mappings.

Hibernate architecture

The Hibernate architecture is modular and includes several components that work together to handle ORM and manage database transactions. Here are the main components:

- **SessionFactory**: A factory for Session objects. SessionFactory is a heavyweight, thread-safe object, created once during the application's startup and used throughout its lifecycle. It holds metadata about the database and mappings and serves as a cache of compiled SQL statements.
- **Session**: Represents a single unit of work with the database. Session is not thread-safe and is used to create, read, and delete persistent objects in a transaction. It wraps a JDBC connection and handles transactions, query execution, and caching.
- **Transaction**: Represents a unit of work within a Session. It handles the atomicity and isolation of data. Transactions ensure data consistency by grouping multiple operations into a single atomic action.
- **Query**: Used to execute database queries in HQL or SQL. Queries can return entities, collections, or individual fields and support parameters, pagination, and caching.
- **Criteria**: Provides a programmatic way to create queries, allowing developers to build queries without using raw HQL or SQL. It is useful for complex queries and offers filtering, projection, and aggregation options.
- **Configuration**: Configures Hibernate by reading mapping

information and database connection properties. Configuration is used to set up Hibernate services, specify database connection details, and map files.

- **Cache**: Hibernate has two levels of caching:
- **First-level cache**: Managed at the Session level, this cache persists for the duration of the session and is only accessible within the same session.
- **Second-level cache**: Managed at the SessionFactory level, this cache persists across sessions and is optional. It allows caching of objects across multiple sessions, improving performance.

Hibernate architecture layers

The following are the hibernate architecture layers:

- **Java application layer**: The application interacts with Hibernate and initiates database operations.
- **Hibernate framework layer**: This layer includes SessionFactory, Session, Transaction, and other ORM mechanisms.
- **Database layer**: The underlying database is where the data resides. Hibernate communicates with the database to fetch and persist objects.

Hibernate mapping types

Hibernate mapping types define how Java objects and their relationships are mapped to database tables. Mapping types are classified into various categories:

- **Basic types (value types)**: These types map simple Java data types directly to database columns. Hibernate provides a wide range of basic types, such as:
    - **Primitive types**: int, float, double, char, etc.
    - **Wrapper types**: Integer, Float, Double, etc.
    - **Date types**: java.util.Date, java.sql.Date, java.sql.Timestamp.
    - **Custom types**: You can define your own custom type

by implementing the UserType interface.

Collection mapping

Hibernate supports various Java collection types, allowing the persistence of collections as fields of an entity. Types of collections in Hibernate:

- **List**: Maps to an ordered list of objects.
- **Set**: Maps to a collection of unique objects.
- **Map**: Maps to a collection of key-value pairs.
- **Bag**: An unordered list that can hold duplicate objects.

Entity relationships mapping

Hibernate supports mapping relationships between entities based on their associations in Java. These associations correspond to relationships in the database and include:

- **One-to-One**: Maps a one-to-one relationship between two entities. It can be implemented by:
- **Primary key association**: Sharing the same primary key between two tables.
- **Foreign key association**: Using a foreign key in one table to refer to the primary key of another table.
- **One-to-many**: Maps a one-to-many relationship where one entity is associated with multiple instances of another entity. This is implemented using a foreign key in the child table that references the primary key of the parent table.
- **Many-to-one**: The inverse of One-to-Many. Each instance of the first entity can be associated with one instance of another entity. In the database, the child table has a foreign key referencing the parent table.
- **Many-to-many**: Maps a many-to-many relationship between two entities. It requires a join table containing foreign keys that reference both tables to handle the association.
- **Component mapping**: Component mapping maps an entity as a value type, meaning a single table may contain both the entity and its component fields. The component class must

be embeddable and represent a value object.

- **Inheritance mapping**: Hibernate supports object-oriented inheritance for mapping class hierarchies to database tables. There are three strategies:
  - **Table per class hierarchy**: Uses a single table for the entire class hierarchy with a discriminator column to differentiate between types.
  - **Table per subclass**: Maps each subclass to its table, including columns for properties inherited from the parent class.
  - **Table per concrete class**: Maps each class in the hierarchy to its table without a shared parent table, duplicating columns for inherited properties across tables.
- **Example of Hibernate mapping in XML and annotations**: In Hibernate, you can define mappings between Java classes and database tables either through XML configuration files or annotations. Below are examples illustrating both approaches.
- **XML mapping example**: XML mapping files define mappings between classes and tables, usually with **.hbm.xml** files:

```
<class name="com.example.Employee" table="employee">
    <id name="id" column="employee_id">
        <generator class="native"/>
    </id>
    <property name="name" column="employee_name"/>
    <property name="salary" column="employee_salary"/>
</class>
```

- **Annotation mapping example**: Annotations can define mappings directly in Java classes:

```
@Entity
@Table(name = "employee")
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private int id;

    @Column(name = "employee_name")
    private String name;

    @Column(name = "employee_salary")
    private double salary;

    // getters and setters
}
```

Hibernate mappings allow you to represent complex relationships and hierarchies, providing a more flexible and object-oriented approach to database operations in Java applications. By reducing boilerplate code and enabling a more object-oriented structure, Hibernate plays a central role in Java-based data persistence.

Hibernate in web frameworks

In a web application, Hibernate fits as the persistence layer (or data access layer). Hibernate is commonly used in the Model layer of a MVC architecture. It manages the application's data and database interactions, making it easier to connect the user interface (View) and application logic (Controller). Hibernate integrates seamlessly with popular Java web frameworks like:

- **Spring Framework**: Hibernate works as the persistence mechanism in Spring-based applications, often using Spring's DAO (Data Access Object) support.
- **Struts**: Hibernate can be integrated with Struts for managing database operations.
- **JavaServer Faces (JSF)**: Hibernate serves as the backend for handling persistent data in JSF applications.
- **Simplifies persistence**: Instead of writing complex JDBC code to manage database connections and SQL statements, Hibernate abstracts these operations, allowing developers to focus on business logic.
- **Transaction management**: Web frameworks often rely on Hibernate's transaction management features to ensure consistency and rollback during failures.

- **RESTful web services**: Hibernate is widely used in RESTful services to manage data persistence. For example, a REST API built using Spring Boot can use Hibernate to handle database operations.

A general workflow in a web framework starts with the user interaction with the UI, sending a request to the controller. The controller calls the service layer to process business logic. The service layer interacts with Hibernate, which translates Java objects to SQL queries and performs database operations. Hibernate manages database interactions seamlessly. The processed data is sent back to the controller and rendered on the UI.

Comparison between the frameworks

Here is a comparative look at some popular Java frameworks—Spring MVC, Hibernate, and Struts—covering key aspects such as architecture, configuration, learning curve, and use cases.

The following are the overview and purpose:

	Purpose
Spring MVC	Spring MVC is designed for building flexible, loosely coupled web applications with a Model-View-Controller architecture.
Hibernate	Hibernate is used for data persistence, allowing developers to map Java classes to database tables and Java objects instead of SQL statements.
Struts	Struts is a mature, MVC-based framework that emphasizes validation and internationalization. It offers a structured approach to building Java web applications.

***Table 15.1**: Comparison between popular Java frameworks based on overview and purpose*

The following is the architecture explained:

	Architecture
Spring MVC	Spring MVC follows the MVC pattern and is highly modular. It consists of a DispatcherServlet as the front controller, with handlers mapped to controller classes, views rendered by JSP or other templating engines, and a model holding application data.
Hibernate	Based on ORM principles, Hibernate maps Java classes to database tables using configuration files or annotations. It uses SessionFactory for managing database sessions and caching, and HQL (Hibernate Query Language) for querying.
Struts	Also based on MVC, Struts relies on an ActionServlet (front controller) that

maps requests to Action classes, which process requests and forward responses to views. Configurations in struts.xml or annotations define the mappings.

***Table 15.2**: Comparison between popular Java frameworks based on architecture*

The following are the core components:

Core Components
Spring MVC	DispatcherServlet, Controllers, View Resolvers, ModelAndView, REST controllers (for REST APIs), Interceptors, Validators, and various configuration files such as applicationContext.xml.
Hibernate	SessionFactory, Session, Transaction, Query, Criteria, Mapping files (.hbm.xml) or JPA annotations, and HQL (Hibernate Query Language).
Struts	Action, ActionForm, ActionServlet, ActionMapping, ActionForward, Interceptors, Validator framework, and configuration files (struts.xml, web.xml).

***Table 15.3**: Comparison between popular Java frameworks based on core components*

The following is the configuration and setup explained:

Configuration
Spring MVC	Spring MVC configuration through XML files (**applicationContext.xml**, **spring-servlet.xml**) or annotations (e.g., **@Controller**, **@RequestMapping**). XML configuration is minimal if using Spring Boot, which simplifies setup significantly.
Hibernate	Needs configuration for database connections and mappings. Configurations can be done in **hibernate.cfg.xml** or programmatically. Entity mapping is done with **.hbm.xml** files or JPA annotations (**@Entity**, **@Table**).
Struts	Relies heavily on XML configuration (struts.xml for action mappings, web.xml for setting up the ActionServlet). Newer versions support annotations (@Action, @Result), which reduce XML dependency.

***Table 15.4**: Comparison between popular Java frameworks based on configuration and setup*

The following table explains the learning curve and documentation:

Learning	Documentation
Spring MVC	Requires knowledge of Spring Core and dependency injection. Spring Boot makes Spring MVC easier to learn by providing pre-configured setups.	Extensive and well-maintained documentation, with many tutorials and examples.
Hibernate	Has a steep learning curve due to its complex configuration options and ORM principles. Understanding lazy loading, caching, and transaction management is crucial.	Good documentation, but finding specific solutions can sometimes be challenging due to the vastness of the framework.
Struts	Relatively easy to learn for basic applications, but mastering its advanced features and customization can be challenging.	Documentation is adequate, but not as comprehensive as Spring's. Community support is also smaller.

***Table 15.5**: Comparison between popular Java frameworks based on learning curve and documentation*

The following is the comparison based on data handling:

	Data handling
Spring MVC	Spring MVC supports form data binding and conversion with @ModelAttribute, making it easy to pass data between views and controllers. It can be integrated with Hibernate or JPA for database operations.
Hibernate	Data is handled as Java objects mapped to relational database tables. Hibernate handles data persistence, retrieval, and caching, simplifying complex database operations.
Struts	Struts has basic data handling with ActionForm classes for form data. It relies on manual database integration or external ORM tools like Hibernate for complex data persistence needs.

***Table 15.6**: Comparison between popular Java frameworks based on data handling*

The following is the comparison based on interceptors and middleware:

	Interceptors
Spring MVC	Spring MVC uses custom interceptors for cross-cutting concerns (logging, authentication, etc.). Configured in XML or programmatically. Filters are also available for request/response modification.
Hibernate	Allows interceptors for operations like auditing, though these are used less frequently. It mainly uses custom event listeners (pre-insert, pre-update, etc.) for entity management.
Struts	Struts relies heavily on interceptors for reusable, cross-cutting logic (e.g., validation, logging). Interceptors are flexible and easy to configure in **struts.xml**.

***Table 15.7**: Comparison between popular Java frameworks based on Interceptor and middleware*

The following is the comparison based on use cases:

	Best suited for
Spring MVC	Highly recommended for RESTful web services, enterprise applications with complex business logic, and applications needing integration with Spring's ecosystem (Spring Boot, Spring Data, Spring Security).
Hibernate	Ideal for applications that need to manage complex data relationships and require robust data persistence. Works well for enterprise applications requiring scalable data storage.
Struts	Suitable for traditional web applications where a straightforward MVC pattern and extensive form handling are needed. Though less popular, Struts is effective in legacy systems and maintenance projects.

***Table 15.8**: Comparison between popular Java frameworks based on use cases*

The following is the comparison based on pros and cons:

	Cons	
Modular MVC design, well-integrated with Spring projects, supports RESTful APIs, dependency injection	Steeper learning curve, requires understanding of Spring ecosystem	
Simplifies data persistence, supports caching, database portability, minimizes SQL usage	Can be complex to configure, not ideal for simple applications	
Established MVC pattern, XML-driven configuration, good for legacy applications	Outdated compared to Spring MVC, less flexible, not suitable for new projects	

***Table 15.9**: Comparison between popular Java frameworks based on pros and cons*

After this comparative analysis, it can be concluded that Spring MVC is a versatile choice for modern web applications, especially those that leverage the Spring ecosystem. Hibernate excels in data persistence and complex database interactions, often used alongside Spring MVC for a complete web application solution. Struts remains valuable for traditional MVC web applications, though it is increasingly being replaced by Spring MVC in new projects.

Choosing the right framework depends on specific project needs. For instance:

- Use Spring MVC if you need a full-fledged enterprise application with REST services.
- Use Hibernate for handling complex database transactions.
- Use Struts when working with or maintaining legacy Java applications that already use this framework.

Conclusion

To conclude this chapter, we started with JavaBean and then we have explored the essential Java web development frameworks—Spring MVC and Struts—that empower developers to create efficient, scalable, and well-organized applications. We have also discussed Hibernate as a robust solution for managing persistence in modern web frameworks, ensuring a clean separation of concerns and efficient data handling.

Each framework has unique strengths tailored for different project

needs: Spring MVC excels with its integration capabilities and REST support, Hibernate offers robust data persistence with ORM features, and Struts provides a traditional MVC structure beneficial in legacy and maintenance projects. By understanding their architectures, core components, and suitable use cases, developers can make informed decisions on framework selection. This foundational knowledge enables the creation of flexible, maintainable web applications adaptable to various business requirements and technological environments.

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Index

A

Abstract Class 115
Abstract Class, characteristics 115
Abstract Class, program 117
Abstract Class, use cases
  Common Base Class 117
  Default, behavior 117
  Partial, implementing 117
  Shared State 117
Arithmetic Operators 16
Arrays 47
Arrays, types
  Multi-Dimensional Array 49
  One-Dimensional Array 47
Assignment Operators 18

B

Binary Large Object (BLOB) 317
Bitwise Operators 22, 24
Bitwise Operators, steps 25, 26
BLOB, configuring 317-319
Break Statement 44
Break Statement, concept
  Inside Loop 44
  Switch 45
Built-in Exceptions 153-155
Buttons 248
Buttons, features
  JButton 248, 249
  JCheckBox 251
  JComboBox 258
  JList 256
  JRadioButton 253
  JTextField 254
  JToggleButton 250
Byte Stream Classes, methods
  DataInputStream 173
  DataOutputStream 172

InputStream 171
OutputStream 171, 172

C

CallableStatement 308, 310
CamelCase 10
Centralize Error Codes 161, 162
Centralize Error Codes, approaches
Fail Fast 163
Handle Exceptions 163
Multi-Catch Blocks 163, 164
Character Large Object (CLOB) 319
Character Stream Classes, methods
BufferedReader 177
FileReader 176
FileWriter 178, 179
InputStreamReader 175, 176
PrintWriter 179-181
Reader 175
Writer 178
Class 56, 57
Class, configuring 57-59
Class, preventing 59-61
Class Value, returning 61, 62
Class With Parameters, optimizing 62-65
CLOB, configuring 319, 320
Common Event Classes 265
Common Event Classes, interfaces
Action Event 265-267
Event Adapters 275
Key Event 274
MouseEvent 270
Window Event 275
Communication Threads 223-226
Communication Threads, scenarios
Bank Account, managing 226, 228
Real-Time Stock Price, monitoring 228
Components 244
Components, methods
Buttons 248
ImageIcon 247
JComponent 244
JLabel 245
Conditional Operators 27, 28
Conditional Statements, types
If-else Ladder 37, 38
If else Statement 35
Multiple If Structure 36

Switch Case Construct 38, 40
Constructor 65, 66
Constructor, architecture 66-68
Constructor, benefits
Automatic Initialization 68
Code, conciseness 68
Object-Oriented Design 68
Constructor, methods 68
Containers 232
Containers, categories
Intermediate Containers 241
Top-Level Containers 232
Continue Statement 45, 46
Control Structures 35
Control Structures, points
Conditional Statements 35
Loops 40
Cookies 388
Cookies, attributes 388, 389
Cookies, configuring 389, 390
Cookies, modifying 390
Cookies, types
Persistent 388
Session 388
Covariant Return Type 110
Covariant Return Type, scenario 110, 112

D

Database Operations 299
Database Operations, steps
Connection, establishing 299, 300
Exceptions, handling 303
JDBC Driver, interacting 299
Resources, closing 302
ResultSet, handling 302
SQL Queries, executing 300, 301
Statement 300
Transaction, handling 301
DatagramPacket 348
DatagramPacket, configuring 348-353
DatagramPacket, methods 348
DataGramSocket 345
DataGramSocket, constructors 345
DataGramSocket, methods 346
destroy() 367
Dynamic Method Dispatch 108, 109

E

Event Handling 264
Event Handling, components
  Event 264
  Event Listener 264
  Event Object 265
  Event Source 264
Event Handling, mechanism
  Event Listener 265
  Event Object 265
  Event Source 265
Exception Class 140
Exception Class, configuring 141, 142
Exception Handling 142
Exception Handling, best practices
  Avoid, swallowing 159
  Catch Specific 159
  Cleanup 159
  Custom Execeptions 160
  Javadoc 161
  Log Exceptions 160
Exception Handling, points
  Finally Clause 152, 153
  Throw Clause 149, 150
  Throws 150-152
  Try/Catch, configuring 143-147
Exception Handling, syntax 142, 143

F

File Input/Output 189
File Input/Output, classes
  BufferedReader 195
  BufferedWriter 196
  Byte Stream 190
  Character Streams 193
  FileReader 193
  FileWriter 194
  OutputStream 191
  PrintWriter, simplifying 197
  Source File, copying 192
final keyword 82
final keyword, contexts
  final class 83
  final method 82
  final variable 82
final keyword, role 112-114

G

Garbage Collection (GC) 74

H

Hibernate Architecture 477
Hibernate Architecture, components 477
Hibernate Architecture, layers 478
Hibernate Framework 477
Hibernate Framework, features
  Automatic Table, generating 477
  Caching 477
  Hibernate Query Language (HQL) 477
  Lazy, loading 477
  Transparent Persistence 477
HTTP, methods
  GET 364
  POST 364
HttpServlet 369-372
HttpServlet, methods 369
HttpServletRequest 372
HttpServletResponse 373-377
HttpSession 382-387
HttpSession, methods 382
Hypertext Transfer Protocol (HTTP) 364

I

Identifier 10
ImageIcon 247
Increment/Decrement Operators 26, 27
InetAddress Class 335
InetAddress Class, configuring 335-338
Inheritance 94
Inheritance, configuring 94-99
Inheritance, types
  Hierarchical 100, 101
  Hybrid 101
  Multilevel 100, 103
  Multiple 102
  Single 99, 100
Input/Output 182
Input/Output, concepts 168
Input/Output, sections
  BufferedReader 184, 185
  Bytes Directly, reading 182, 183
  Console, printing 187, 188
  Error Message, handling 189
  InputStreamReader 183, 184

Scanner Class 186
User Data, reading 182
Input/Output, streams
System.err 182
System.in 182
System.out 182
Interface 122
Interface/Abstract Class, comparing 126, 127
Interface, architecture 122, 123
Interface, benefits
Abstraction 126
Code, reusability 126
Loose, coupling 126
Multiple, inheritance 126
Interface, extending 127-129
Interface, implementing 123-126
Intermediate Containers 241
Intermediate Containers, types
JPanel 241
JScrollPane 241, 242
JSplitPane 243, 244
JTabbedPane 242, 243

J

J2EE, architecture
N-Tier 362, 363
Single-Tier 361
Three-Tier 362
Two-Tier 361
JApplet 239
JApplet, implementing 240, 241
JApplet, methods 239
Java 2 Enterprise Edition (J2EE) 360
Java Architecture 6, 7
Java Architecture, components
Java Development Kit (JDK) 7
Java Runtime Environment (JRE) 7
Java Virtual Machine (JVM) 7
JavaBeans 458
JavaBeans, benefits
Encapsulation 459
Interoperability 459
Reusability 459
JavaBeans, features 458
Java/C++, difference 4, 5
Java, features
Automatic Memory, managing 3
Community/Ecosystem 4

Familiar, syntax 3
Multi-Thread, supporting 4
Object-Oriented 3
platform, independence 3
Portability 3
Rich Standard, library 4
Robustness 3
Security 3
Java, history 2, 3
Java Program 30
Java Program, configuring 33-35
Java Program, executing 31
Java Program, structure 31, 32
JavaServer Faces (JSF) 434
Java/SQL Data, mapping 315, 316
Java Tokens 8
Java Tokens, fundamentals
Data Types 9
Keywords 8
Literals 9
Java Web Frameworks 463
Java Web Frameworks, aspects 481
Java Web Frameworks, purpose
Efficiency 463
Scalability 463
Security 463
Standardization 463
Java Web Frameworks, types
Spring MVC 463
Struts Architecture 473
Struts Framework 471
JComponent 244
JDBC 282, 283
JDBC, advantages
Database, independence 283
Flexibility 284
Portability 283
standardization 283
JDBC Application, creating 303, 305
JDBC Classes 288
JDBC Classes, interfaces
CallableStatement 294
Connection 290
DriverManager 289
PreparedStatement 292
ResultSet 296
SQLException 298
Statement 291, 292

JDBC Drivers, types
  JDBC-ODBC Bridge 284, 285
  Native-API Driver 285, 286
  Network Protocol Driver 286, 287
  Thin Driver 287, 288
JDBC, layers
  Application Layer 283
  Driver Layer 283
JDBC With Swing
, analyzing 312, 315
JDialog 238
JDialog, constructors 239
JDialog, methods 239
JDialog, types
  Application Modal 238
  Document Modal 238
  Modeless 238
  Toolkit Modal 238
JFrame 233, 234
JFrame, constructors 234
JFrame, features 235, 236
JFrame, methods 234
JLabel 245
JSF/Angular, comparing 437, 438
JSF/Apache Struts, comparing 436
JSF Application, building 444-447
JSF, features 434, 435
JSF/Spring MVC, comparing 435, 436
JSF Web Application 439
JSF Web Application, architecture
  MVC Architecture 439
  Request Processing 442
JSP 402
JSP, architecture 405
JSP, components 405, 406
JSP, features 402, 403
JSP Implicit Objects 418
JSP Implicit Objects, purpose
  application 419, 420
  config 420
  out 419
  request 418
  response 418
  session 419
JSP Life Cycle 406
JSP Life Cycle, methods
  jspDestroy() 407, 408
  jsplnit() 407

_jspService() 407
JSP Standard Tag Library (JSTL) 423
JSP, syntax
Actions 413, 414
Comments 409
Directives 410-412
Elements, scripting 409, 410
JavaBeans 414-416
JSP With Servlet, integrating 420-423
JSTL, demonstrates 424, 425
JSTL, tags 423
JWindow 236
JWindow, constructors 236
JWindow, implementing 237
JWindow, methods 236
JWindow, use cases 238

L

Layout Managers 260
Layout Managers, strategy
BorderLayout 261
CardLayout 262
FlowLayout 260, 261
GridBagLayout 263
GridLayout 262
lockObject 219
Logical Operators 21
Loops, types
Do-While 42
For 40, 41
Nesting 43
While 41

M

Main Thread 204, 205
Main Thread, illustrating 205, 206
Method Overloading 71, 106
Method Overloading, architecture 71-74
Method Overloading, benefits
Code, maintenance 71
Code, readability 71
Flexibility 71
Method Overloading, concepts
Access Level 106
Constructors 106
Final Methods 106
Instance Methods 106

Method Signature 106
Return Type 106
Method Overloading, points
Code, reusability 108
Extensibility 107
Runtime Polymorphism 107
Multi-Dimensional Array 49
Multi-Dimensional Array, declaring 50
Multi-Dimensional Array, initializing 50, 51
Multithreading 202, 203
MVC Architecture, components
Event Handling 441
Facelets View 440
FacesServlet 440
JSF UI 440
Managed Beans 440
Managed Bean Scopes 441
Navigation 440
Validation/Conversion 441

N

Network 324, 325
Network, concepts
Client/Server, computing 326
IP Address 325
Ports 325, 326
N-Tier 362
N-Tier, types
Business 363
Client 363
Data 363
Data Access 363
Integration 363
Presentation 363

O

Object 56
Object Access Modifiers 69
Object Access Modifiers, types
Default Access 69, 70
Private Access 69
Protected Access 70
Public Access 70, 71
One-Dimensional Array 47
One-Dimensional Array, fundamental
Declaration/
Initialization 47, 48

Element, iterating 48, 49
Elements, accessing 48
Length Property 48
openStream() 332
Operators 15
Operators, types
Arithmetic Operators 16, 17
Assignment Operators 18, 19
Bitwise Operators 22, 23
Conditional Operators 27
Increment/Decrement Operators 26
Logical Operators 21, 22
Relational Operators 19-21

P

Package Naming Conventions 133-137
Packages 129
Packages, architecture 129, 130
Packages, importing 132
Packages, steps 131
PersonBean Application, preventing 462
PersonBean, configuring 459-461
Polymorphism 107
Polymorphism, types 107
PreparedStatement 305, 307
printTable 221

R

Relational Operators 19
Request Processing, phases 442, 443
ResultSet 310
ResultSet, constants 310, 311
ResultSet, operations
Concurrency Constants 311
Holdability Constants 311
Return Statement 46
Runnable Interface 204
RuntimeException 141

S

Scope, concepts
Instance Variables 12, 13
Local Scope 11, 12
Static Variables 13, 14
ServerSocket 338
ServerSocket Clients, handling 341, 342

ServerSocket, constructors 339
ServerSocket, methods 339
Service Method 377, 378
Servlet 366
Servlet API 368
Servlet API, interfaces 368
ServletConfig 396-399
ServletContext 391, 392
ServletContext, methods 392
ServletContext, use cases
  Configuration Parameters 394-396
  Data Across, sharing 393, 394
Servlet, phases
  Initialization 366
  Request, handling 367
Session Management 425
Session Management, points
  Cookies 426, 427
  Hidden From Fields 428
  HTTP Sessions 426
  URL, rewriting 427, 428
Socket 326, 327
Socket Address 328
Socket API 327
Socket Class 343
Socket Class, methods 343
Socket, rules
  Internet Protocol (IP) 327
  TCP 327
  UDP 327
Socket, types
  TCP 328
  UDP 328
Software Development 5
Software Development, domains
  AI/ML 6
  Big Data, analytics 6
  Cloud, computing 6
  Enterprise Applications 5
  Internet of Things (IoT) 6
  Mobile Development 6
  Web Development 6
Spring MVC
  about 463
  Advantages 469
  architecture 463, 464
  Exception, handling 469
  Module, optimizing 465

Request Workflow, handling 468
Spring API 467
Spring Bean, implementing 466
Spring Boot, preventing 470
State Management 381, 382
State Management, fields
Cookies 382
Hidden Form 382
URL, rewriting 382
static keyword 78
static keyword, sections
static blocks 80, 81
static methods 79
static variables 79
Streams 168, 169
Streams, types
Byte Stream Classes 169, 170
Character Stream Classes 174
String Class 86
String Class, methods 86, 87, 88
String Class, ways 86
Struts Architecture, components
Controller 473
Model 473
View 473
Struts Architecture, points
Core Interceptors 474
Custom Interceptors 474
Interceptors 473
Interceptor Stack 475
Request Flow 473
Validator Framework 475
Struts Framework 471
Struts Framework, components
Action 472
ActionForm 471
ActionForward 472
ActionMapping 472
ActionServlet 471
Struts-Config.xml 472
Struts Framework, features 471
Struts Framework, files 472
super keyword 103
super keyword, functionalities
Superclass Constructors 104, 105
Superclass Fields, accessing 104
Superclass Methods, accessing 103, 104
Synchronization 216

Synchronization, scenarios
Atomicity Problems 216
Race Conditions 216
Visibility Issues 216
Synchronized Method 216, 218
Synchronized Statements 218-220

T

this keyword 74
this keyword, uses 74-77
Thread Class 203
Thread Class, extending 206
Thread Class, lifecycle 210, 211
Thread Class, methods 203-214
Thread Class, preventing 208-210
Thread Class, states
Blocked 210
New 210
Runnable 210
Terminated 210
Timed, waiting 210
Waiting 210
Thread With Runnable Interface, optimizing 207
Top-Level Containers 232
Top-Level Containers, layers
Content Pane 233
Glass Pane 233
Layered Pane 233
Optional Menu Bar 233
Root Pane 232
Top-Level Containers, structures
JApplet 239
JDialog 238
JFrame 233
JWindow 236
totalMarks() 61
try-with-resources 198
try-with-resources, benefits 200
try-with-resources, managing 198, 199
try-with-resources, preventing 199

U

Uniform Resource Locator (URL) 329
URL, components
Protocol Identifier 329
Resource Name 329
URL, configuring 332, 333

URLConnection 333-335
URL, steps
  creating 330, 331
  Parsing 331, 332
User-Defined Exceptions 157, 158

V

Variables 10
Variables, concepts
  Lifetime 14
  Scope 11

W

Web Application 363
Web Container 365
Web Container, responsibilities 365
Web Server 365
Wrapper Classes 84
Wrapper Classes, configuring 84, 85
Wrapper Classes, types 84